НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА СИСТЕМНОГО ПРОГРАМУВАННЯ І
СПЕЦІАЛІЗОВАНИХ КОМП'ЮТЕРНИХ СИСТЕМ

«На правах рукопису»
УДК 044.77

«До захисту допущено»
Завідувач кафедри СПСКС

_____ В.П.Тарасенко_
(підпис) (ініціали, прізвище)

"___"_____2018р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 123 Комп'ютерна інженерія
Спеціалізовані комп'ютерні системи

на тему: Спосіб порівняння графів потоку керування програми

Виконав: студент II курсу, групи КВ-73мп

Грек Олександр Васильович

Науковий керівник доцент кафедри, ктн, Марченко О.І.

Рецензент _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2018 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп'ютерних систем

Рівень вищої освіти – другий (магістерський)
Спеціальність 123 Комп'ютерна інженерія
Спеціалізовані комп'ютерні системи

ЗАТВЕРДЖУЮ
Завідувач кафедри СПСКС
_____ _В.П.Тарасенко__
(підпис) (ініціали, прізвище)
«___»_____2018р.

ЗАВДАННЯ

на магістерську дисертацію студенту
Греку Олександру Васильовичу

1. Тема дисертації Спосіб порівняння графів потоку керування програми, науковий керівник дисертації Марченко Олександр Іванович доцент кафедри СПіСКС, к.т.н., доцент

затверджені наказом по університету від «__» _____ 2018 р. №___

2. Термін подання студентом дисертації 07 грудня 2018 р.

3. Об'єкт дослідження – підвищення ефективності порівняння графів потоку керування з метою пошуку дублювання коду.

4. Предмет дослідження – способи пошуку дублювання коду за допомогою визначення ізоморфізму графів.

5. Перелік завдань, які потрібно розробити
   - опис предметної області досліджень та обґрунтування способу порівняння графів потоку керування програми;
   - спосіб побудови та зберігання графів потоку керування програми;
   - спосіб порівняння побудованих графів потоку керування програми.

6. Перелік ілюстративного матеріалу
   - Блок-схема модифікованого алгоритму побудови графів потоку керування.
   - Блок-схема алгоритму порівняння графів потоку керування.
   - Діаграма класів.
   - Таблиця результатів
   - Використані формули
7. Перелік публікацій
   - «Підвищення точності порівняння програм», міжнародна науково-практична конференція «Цілі сталого розвитку третього тисячоліття». – 2018;
   - «Спосіб порівняння графів потоку керування програми», XI конференція молодих вчених ПМК-2018-2. – 2018;

8. Дата видачі завдання 5 вересня 2017 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Вивчення літератури за тематикою проекту	05.09.2017	
2	Аналіз існуючих рішень	20.01.2018	
3	Підготовка матеріалів першого розділу магістерської дисертації	09.03.2018	
4	Підготовка матеріалів другого розділу магістерської дисертації	30.04.2018	
5	Підготовка матеріалів третього розділу магістерської дисертації	10.09.2018	
6	Підготовка графічної частини дипломного проекту	16.10.2018	
7	Оформлення документації дипломного проекту	01.11.2018	
8	Попередній розгляд магістерської дисертації на кафедрі	26.11.2018	

Студент _____ Грек О.В.

Науковий керівник дисертації _____ Марченко О.І.

Актуальність теми. Створення та підтримка великих програмних систем - це нагальна проблема для сучасного бізнесу у сфері інформаційних технологій. З метою автоматичного виявлення помилок у програмному коді все більше використовуються системи статичного аналізу коду. Відсутність дублювання коду вважається критерієм якості коду, так як копіювання вихідного тексту програми може призвести до виникнення помилок, що не можуть бути ефективно знайдені автоматизованими засобами. Для вирішення задачі пошуку дублікатів у програмному коді використовується як текстовий аналіз вихідного коду, так і аналіз за допомогою метрик та графів. Останній дає більш високу точність, але низьку швидкість роботи. Тому створення способів, що дозволяють прискорити визначення дублювання коду за допомогою порівняння графів потоку керування програми є актуальним.

Об'єктом дослідження є процес визначення співпадіння графів потоку керування програми.

Предметом дослідження є способи визначення дублювання коду за допомогою порівняння графів потоку керування програми.

Мета роботи: прискорення процесу визначення співпадіння графів потоку керування, розробка більш ефективного способу зберігання графів потоку виконання для прискорення процесу порівняння.

Наукова новизна:

1. Проаналізовано існуючі системи та способи визначення дублювання коду і показано, що ці системи мають недоліки у їх використанні за різними показниками: недостатня точність визначення при змінах вхідних даних, низька швидкість роботи та обмеженість варіантів використання.

2. Запропоновано спосіб зберігання графів потоку керування програми для визначення відповідності графів, який відрізняється від інших тим, що завдяки використанню спеціального стисненого представлення графів дозволяє зменшити використання дискового простору для зберігання графів.

3. Запропоновано спосіб побудови та порівняння графів потоку керування програми для визначення відповідності дерев, який відрізняється від інших тим, що відповідність підграфів двох графів визначається на основі співпадіння їх текстового представлення, що дозволяє прискорити процес порівняння для кодової бази великого об'єму.

4. Виконано порівняльний аналіз розробленого способу з існуючими аналогами для вирішення задачі визначення співпадіння графів потоку керування у вихідному коді мовою програмування C# і показано, що модифікований спосіб має вищі показники швидкості роботи, але нижчі показники точності визначення.

Практична цінність отриманих в роботі результатів полягає в тому, що розроблений спосіб дозволяє прискорити процес пошуку дублікатів програмного коду у випадку регулярного використання, якщо побудову графів потоку виконання необхідно виконати лише для зміненої частини коду. Крім того, запропонований спосіб може бути використаний для контролю ліцензій контенту у публічних репозиторіях. Система, що використовувалась для тестування розробленого способу, може використовуватись як модуль аналізу коду у системі збірки MSBuild.

Апробація роботи. Система для визначення шаблонів проектування у програмах

була представлена та обговорювалась на науковій конференції магістрантів та аспірантів “Прикладна математика та комп’ютинг” ПМК-2018-2 (Київ, 15 листопада 2018 р.), «Підвищення точності порівняння програм», міжнародній науково-практичній конференції «Цілі сталого розвитку третього тисячоліття» (Київ, 23-25 травня 2018 р.), IV Міжнародній науково-технічній Internet-конференції «Сучасні методи, інформаційне, програмне та технічне забезпечення систем керування організаційно-технічними та технологічними комплексами» (Київ, 22 листопада 2018 р.).

Структура та обсяг роботи. Магістерська дисертація складається з вступу, чотирьох розділів та висновків.

У вступі подано узагальнену оцінку сучасного стану проблеми, обґрунтовано актуальність виконаного дослідження, дано загальну характеристику роботи, поставлено мету та задачу дослідження, і наведено практичну цінність роботи.

У першому розділі розглянуто існуючі системи для визначення дублювання коду у програмах, їх класифікацію та особливості, недоліки та переваги кожного сопобу. Розглянуто різні існуючі програмні системи, що вирішують дану проблему.

У другому розділі розглянуто способи побудови, зберігання та виявлення подібності графів потоку керування програми. Запропоновано способи зберігання, а також побудови та порівняння графів потоку керування програми для пошуку дублікатів програмного коду.

У третьому розділі описано особливості реалізації розробленої системи.

У четвертому розділі наведено методику тестування системи та порівняння розробленої системи з аналогами.

У висновках представлені результати проведеної роботи.

Робота представлена на * аркушах, містить посилання на список використаних літературних джерел.

Ключові слова: виявлення дублювання коду, граф потоку керування, подібність графів.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

.NET Framework – фреймворк від компанії Microsoft, що включає віртуальну машину, а також необхідні інструменти для розробки та виконання як користувацьких застосунків, так і серверних програм.

CLR – віртуальна машина фреймворку Microsoft .NET Framework.

PowerShell – програмна оболонка командного рядка, що використовується у операційних системах сімейства Microsoft Windows.

Граф залежності програми – представлення програми у вигляді графу, що показує залежності виконання кожної операції від даних та керуючих конструкцій для кожної операції [1].

Граф потоку керування – це множина усіх можливих шляхів виконання програми, що представлена у вигляді графу [2].

Лексема – синтаксична одиниця, що відповідає певній послідовності визначених символів та оброблюється на етапі лексичного аналізу.

Рефакторинг – процес внесення структурних змін у програмний код, що не змінюють поведінку кінцевої системи.

Серіалізація – трансформація об'єктів у пам'яті програмної системи у визначений формат з метою їх зберігання або передачі.

Синтаксичне дерево – є однією із внутрішніх форм подання вхідної програми у вигляді дерева, що містить структурні одиниці мови, а також зв’язки між ними [3].

Фреймворк – у програмній інженерії це абстракція, у якій програмне забезпечення з узагальненою функціональністю може бути вибірково змінене за допомогою коду, написаного користувачем.

ВСТУП

Дублювання програмного коду є значною проблемою на даному етапі розвитку індустрії програмного забезпечення. Автоматизація процесу пошуку дублювання коду допомагає знайти місця потенційних помилок у програмному коді, визначити джерело коду та авторство програми. Своєчасний аналіз коду на подібність може призвести до зменшення витрат часу на виправлення помилок та до прийняття правильних рішень щодо змін архітектури вихідного коду. Особливо це актуально для великих за обсягом проектів, які підтримуються командою розробників.

Складність виконання такої задачі полягає в тому, що дублікати вихідного коду не обов'язково є ідентичними з точки зору вихідного тексту. В загальному випадку два фрагменти коду, що виконують той самий алгоритм тим самим способом, є дублікатами, незалежно від мови програмування, використаних мовних конструкцій та іменування ідентифікаторів. Для вирішення цієї задачі використовують як текстове порівняння вихідного коду, так і порівняння синтаксичних структур програми, її метрик та графів.

Існуючі інструменти, що призначені для пошуку дублювання програмного коду, поділяються на дві категорії: текстові та семантичні. Текстові можуть використовуватись лише в межах однієї мови програмування та не забезпечують належної якості порівняння у випадку незначних змін вихідного коду, проте мають високу швидкість роботи. Семантичні виконують глибокий аналіз програми, що забезпечує високу точність порівняння, але їх ефективність значно знижується на великих обсягах коду. Таким чином існує необхідність у створенні інструментів, що забезпечують високу швидкість роботи та не є чутливими до змін вихідного тексту програми.

Основним методом вирішення такої задачі є порівняння потоку керування програми. Дана магістерська дисертація присвячена вирішенню

задачі прискорення процесу порівняння графів та розробці більш швидкого, але менш точного способу порівняння графів, ніж існуючі способи.

1. АНАЛІЗ СПОСОБІВ ПОРІВНЯННЯ ГРАФІВ ПОТОКУ КЕРУВАННЯ ПРОГРАМИ

1.1. Аналіз способів порівняння програмного коду

Дублікат програмного коду – це послідовність програмних інструкцій, присутніх у вихідному коді програми більше одного разу. Довжина такої послідовності має перевищувати задане мінімальне значення. В загальному випадку, дублікатами є будь-які семантичні одиниці програми, що реалізують один алгоритм однаковим способом, але на практиці дублікати визначаються саме за рахунок порівняння послідовностей певних елементів, у вигляді яких представляється вихідний код програми.

Порівняння програмного коду – це задача розрахунку індексу подібності для деякої семантичної одиниці програмного коду. Цією одиницею може бути метод або функція, клас або модуль, файл вихідного коду або програма в цілому. Найбільш зручними варіантами є файл (для будь-яких даних у текстових форматах) та метод (для мов об’єктно-орієнтованого програмування).

Задача пошуку дублікатів зводиться до пошуку максимальних послідовностей синтаксичних конструкцій у вихідному коді програми, для яких індекс подібності перевищує задане граничне значення.

Існуючі методи, що призначені для пошуку дублювання програмного коду, умовно можна розділити на два класи: текстові та семантичні. Методи, що відносяться до текстових, виконують аналіз на рівні тексту вихідного коду програми та не розрізняють синтаксичних конструкцій мови програмування. Порівняння виконується посимвольно, за елементарну одиницю порівняння приймається текстовий рядок. Методи даного класу знаходять послідовності однакових рядків. Дані методи

використовуються не лише у системах пошуку дублювання коду, а і у системах контролю версій та у програмах порівняння текстових файлів.

Текстові методи мають наступні переваги:

- найбільша можлива швидкість роботи;
- незалежність від мови програмування;
- можливість порівняння будь-яких текстових даних.

За рахунок того, що методи прямого текстового порівняння не передбачають виконання лексичного та синтаксичного розбору вхідних даних, тому їх універсальність та швидкість роботи не може бути досягнута будь-якими іншими методами.

Серед недоліків можна виділити те, що текстові методи можуть використовуватись лише в межах однієї мови програмування. Навіть у межах однієї мови програмування якість розпізнавання дублікатів може бути нижча при використанні різних стилів написання коду (рис. 1.1).

```
static void A()
{
    bool dd = true;
    bool dr = false;
    while (dd)
    {
        if (dr)
            yes();
        else
            no();
    }
}
```

```
static void B() {
    bool dd = true;
    bool dr = false;
    while (dd) {
        if (dr)
        {
            yes();
        }
        else { no(); }
        }
    }
```

Рисунок 1.1 – Дублікати коду з використанням різних стилів написання коду

Наявність коментарів, використання різних синтаксичних конструкцій, зміна значень констант та іменування ідентифікаторів, використання відступів та символів табуляції – це фактори, що значно знижують точність розпізнавання дублікатів при використанні текстових методів розпізнавання дублікатів. Можна зробити висновок, що способи пошуку дублікатів програмного коду, що базуються на аналізі текстового

представлення коду, не забезпечують належної якості порівняння у випадку незначних змін вихідного коду, проте мають високу швидкість роботи.

Семантичні методи виконують більш глибокий аналіз вихідного коду програми, що забезпечує кращу точність визначення дублікатів, у порівнянні з текстовими методами, проте їх ефективність значно знижується на великих обсягах коду.

Способи, що базуються на порівнянні вихідного тексту програмного коду, зазвичай виконують порівняння послідовно для кожного рядку коду, і якщо послідовність рядків одного файлу відповідає послідовності рядків іншого файлу, то частини файлів позначаються як підозрілі. Різні алгоритми використовують різні способи числового розрахунку індексу подібності, а деякі інструменти не розраховують його взагалі.

Більш точні способи порівняння потребують більшої кількості кроків, тому вони не можуть гарантувати таку ж швидкість, як способи текстового порівняння. Початкові кроки їх роботи подібні роботі сучасного компілятора [4]. Може виконуватись наступна послідовність кроків:

- лексичний аналіз;
- синтаксичний аналіз;
- побудова графу потоку керування;
- аналіз потоку даних;
- побудова графу залежностей програми;
- порівняння елементів програми у заданому представленні.

Лексичний аналіз – це перший крок, що є необхідним для будь-яких методів пошуку дублікатів, крім текстових, за умови використання мови програмування, лексика якої відома [5]. Текст вихідного коду програми розбивається на лексеми відповідно до лексичної частини граматики мови програмування (рис. 1.2). Більшість інструментів виконують порівняння саме на рівні лексем (за умови наявності лексичного аналізатора для даної

мови, інакше – на рівні символів) та не виконують наступних кроків, крім останнього. Лексичний аналіз не потребує значних ресурсів та може виконуватись за один прохід, таким чином незначне зменшення ефективності дозволяє значно підвищити точність визначення за рахунок можливості використання таких покращень, як ігнорування коментарів, імен ідентифікаторів та константних значень [6].

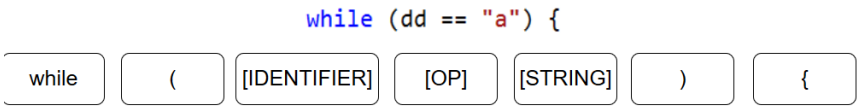


Рисунок 1.2 – Представлення коду у вигляді послідовності лексем

На етапі синтаксичного аналізу за допомогою набору лексем, отриманого у попередньому кроці, будується синтаксичне дерево відповідно до граматики мови програмування (рис. 1.3). Залежно від конкретного алгоритму, може виконуватись конвертація дерева у абстрактне синтаксичне дерево. Алгоритми пошуку дублікатів, що виконують порівняння на рівні синтаксичних дерев, можуть використовувати способи оптимізації, які дозволяє мова програмування, наприклад, ігнорування виразів, ігнорування типів циклічних інструкцій тощо. Проте для цього необхідно мати повноцінний синтаксичний аналізатор певної мови програмування та виконувати досить складний процес синтаксичного аналізу для кожного файлу. При роботі з великим об’ємом коду, що необхідно проаналізувати на подібність, виникає проблема ефективного зберігання дерев у пам’яті.

За допомогою обходу синтаксичного дерева в глибину виконується побудова графу потоку керування, після чого в деяких алгоритмах розраховуються певні метрики коду, такі як цикломатична складність та метрика Халстеда. Представлення програми у вигляді графу потоку керування (рис. 1.4) дозволяє виконувати еквівалентні перетворення, завдяки яким можна визначити подібність програм при використанні різних синтаксичних конструкцій.

```
while (dd == "a")
{
    dd = dd.Substring(1);
    i++;
}
```

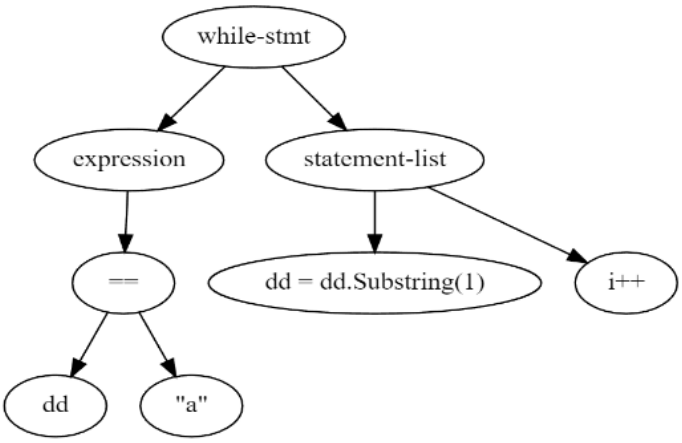


Рисунок 1.3 – Представлення коду у вигляді абстрактного синтаксичного дерева

Процес побудови та можливі варіанти представлення графу потоку керування будуть описані нижче. Існують методи порівняння, що приймають попереднє рішення про наявність дублікатів за допомогою порівняння розрахованих метрик коду. Для таких методів важливо використовувати найбільшу можливу кількість показників, що оцінюють різні аспекти коду. Порівняння коду на рівні потоку керування програми та на рівні розрахованих числових показників не використовується у популярних сучасних інструментах для пошуку дублювання коду.

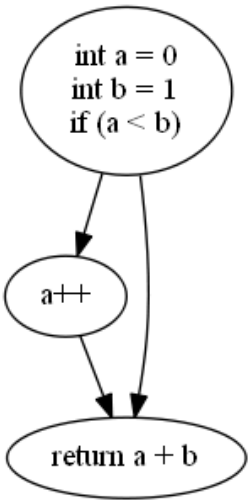


Рисунок 1.4 – Граф потоку керування для умовної інструкції

Аналіз потоку даних дозволяє визначити, які значення можуть приймати змінні у кожній точці виконання програми. Для виконання аналізу потоку даних найчастіше використовується зворотній обхід графу

потоку керування та семантичний аналіз його вершин. Це досить складна операція, тому виконання даного виду аналізу займає досить багато часу та знижує швидкість алгоритмів, що на ньому базуються. Аналіз потоку даних використовується для побудови графу залежності для програми. Графи залежності програми (рис. 1.5) використовуються у системах пошуку дублювання коду та пошуку плагіату, а також у антивірусних системах як найбільш точний та надійний спосіб порівняння програмного коду, проте він виконується достатньо довго, так як вимагає виконання усіх попередніх кроків [7].

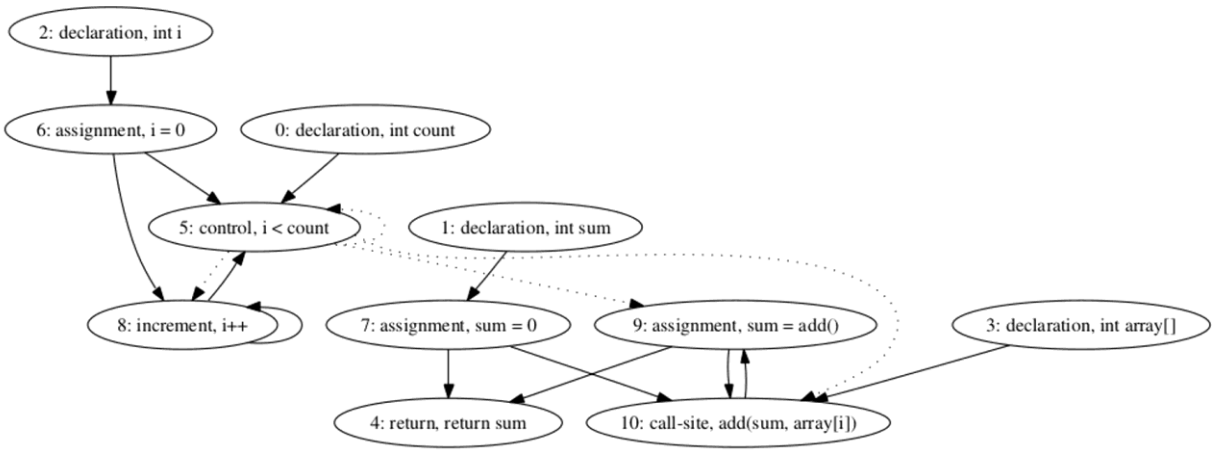


Рисунок 1.5 – Граф залежності програми

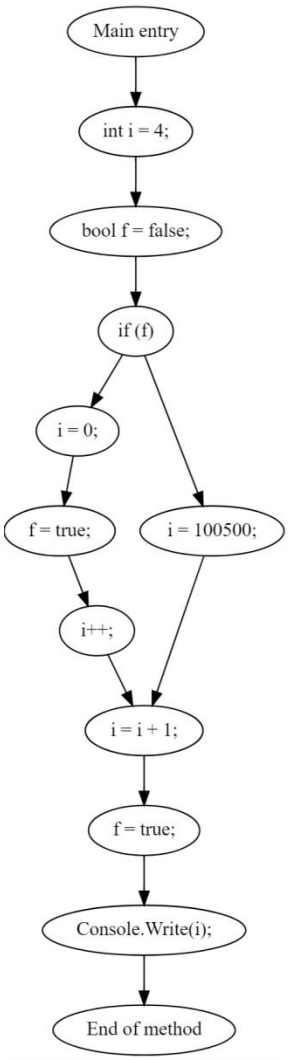
Останнім кроком будь-якого методу пошуку дублікатів є порівняння – процес пошуку дублікатів у побудованому представленні програми. Це може бути порівняння послідовностей лексем, порівняння синтаксичних дерев, порівняння розрахованих метрик коду, порівняння графів потоку управління або графів залежності програми. Кінцевим результатом порівняння має бути визначення підозрілих місць у вихідному коді програми або розрахунок індексу подібності.

1.2. Графи потоку керування та їх порівняння

Граф потоку керування – представлення усіх можливих шляхів виконання програми у вигляді орієнтованого графу (рис. 1.6). Вершинами

графу є блоки інструкцій, що не містять інструкцій передачі керування. Ребра відповідають можливим шляхам між цими блоками [7, 8].

Графи потоку керування широко використовуються у інформаційних технологіях. Дані графи використовуються для рефакторингу та візуалізації коду, а також для пошуку дублювання коду. Граф потоку керування є достатньо зручним завдяки простоті та ефективності побудови. Сучасні інтегровані середовища розробки виконують аналіз на графах потоку керування у режимі реального часу для визначення недоступного коду та інших діагностик.




```
static void Main(string[] args)
{
    int i = 4;
    bool f = false;
    if (f)
    {
        i = 0;
        f = true;
        i++;
    }
    else
    {
        i = 100500;
    }
    i = i + 1;
    f = true;
    Console.Write(i);
}
```

Рисунок 1.6 – Граф потоку керування для методу

Порівняння графів потоку керування – це задача пошуку максимальних підграфів двох графів, що є ізоморфними, при чому

додавання будь-якого вузла до кожного підграфу зробить їх не ізоморфними [8].

Постає задача визначення спільних вузлів. Так як вузол звичайного графу потоку керування складається з декількох синтаксичних інструкцій, що виконуються послідовно, визначення відповідності вузлів є складним завданням, що не має однозначного правильного рішення. У більшості теоретичних досліджень використовуються наступні методи визначення подібності вузлів:

- прямий перебір – якщо усі елементи першого вузла є елементами другого вузла, та порядок елементів зберігається – вузли є відповідними;
- повний збіг – якщо усі елементи першого вузла є елементами другого вузла, їх порядок та кількість збігається – вузли є відповідними;
- розрахунок індексу подібності – для пари елементів розраховується індекс подібності за наступною формулою (1):

$$i = \frac{2S(x_1 x_2)}{2S(x_1 x_2)+U(x_1)+U(x_2)} \quad (1)$$

У формулі S – кількість спільних елементів у двох вузлах, $U(x_1)$ – кількість унікальних елементів у першому вузлі, $U(x_2)$ – у другому вузлі. Елементи вважаються спільними тільки у тому випадку, якщо їх порядок збігається. Імена ідентифікаторів, значення констант та особливості форматування коду не враховуються при порівнянні [9].

1.3. Аналіз існуючих засобів пошуку дублікатів програмного коду

Аналіз існуючих засобів пошуку дублікатів коду необхідно проводити за двома характеристиками: швидкістю роботи та якістю визначення дублікатів. Час роботи визначається за допомогою таймера комп’ютера.

Існуючі дослідження в області пошуку дублювання коду поділяють дублікати на чотири основні типи:

- тип 1: дублікати, що відрізняються лише за стилем написання коду (рис. 1.1);
- тип 2: дублікати, що відрізняються іменами змінних та значеннями констант (рис. 1.7);
- тип 3: дублікати, що містять додані або змінені інструкції (рис. 1.8);
- тип 4: дублікати, що не є синтаксично подібними, але є функціонально подібними – виконують ті самі функції (рис. 1.9).

```
static void Ax()
{
    string some_str = "unknown";
    bool some_bool = true; //bool
    while (some_str == /* "a" */ "b")
    {
        {
            if (some_bool)
                yes_func();
            else
                no_func();
        }
    }
}
```

```
static void A()
{
    string dd = "true";
    bool dr = false;
    while (dd == "a")
    {
        {
            if (dr)
                yes();
            else
                no();
        }
    }
}
```

Рисунок 1.7 – Дублікати другого типу

Дублікати другого типу можуть бути розпізнані на етапі порівняння лексем. Якщо лексема є ідентифікатором або значенням константи, її реальне значення ігнорується. Опція ігнорування імен та значень присутня в усіх сучасних засобах пошуку дублікатів, в більшості розглянутих систем вона не може бути вимкнена [10].

Дублікати четвертого типу також називають семантичними дублікатами. Наразі не існує способу визначення дублікатів четвертого типу без використання глибокого семантичного аналізу та часткової

емуляції, тому сучасні інструменти не можуть використовуватись для пошуку дублікатів коду такого типу.

```
static void A()
{
    string dd = "true";
    bool dr = false;
    while (dd == "a")
    {
        {
            if (dr)
                yes();
            else
                no();
        }
    }
}
```

```
static void A3()
{
    bool dr = false;
    var some = "true";
    while (!("a" != some))
    {
        {
            if (dr)
            {
                yes();
                A();
            }
            else
                no();
        }
    }
    A();
}
```

Рисунок 1.8 – Дублікати третього типу

Задача пошуку таких дублікатів частково вирішена в області пошуку поліморфних комп'ютерних вірусів, а у сфері пошуку дублікатів вихідного коду дублікати четвертого типу не розглядаються [11]. Деякі дублікати четвертого типу можуть бути розпізнані за допомогою порівняння графів залежності програми.

```
int main() {
    int a = 2;
    int b = a + 9;
    return b;
}
```

```
int add()
{
    int n = 10;
    return ++n;
}
```

Рисунок 1.9 – Дублікати четвертого типу

Теоретична можливість розпізнавання дублікатів кожного типу представлена у таблиці 1.1.

Таблиця 1.1 – Відповідність класів порівняння та типів дублікатів, що можуть біти знайдені за допомогою даних класів

Клас порівняння	**Тип 1**	**Тип 2**	**Тип 3**	**Тип 4**
Текст	Частково	Ні	Ні	Ні
Лексеми	Так	Так	Ні	Ні
Метрики	Так	Так	Частково	Частково
Синтаксичні дерева	Так	Так	Частково	Ні
Графи	Так	Так	Так	Частково

У таблиці 1.2 наведено набір протестованих програм та їх клас порівняння. Кожна з наведених програм відповідає своєму класу та може якісно розпізнавати дублікати коду, що відповідають її класу [12].

Таблиця 1.2 – Існуючі інструменти пошуку дублікатів коду

Найменування	**Клас порівняння**
Duplo(c)	Текст
NiCad4	Текст
Simian	Текст
SDD	Текст
CCFinder(X)	Лексеми
CPD	Лексеми
CP-Miner	Метрики
CloneDigger	Синтаксичні дерева
CloneDR	Синтаксичні дерева
Scorpio	Графи залежності

Наразі не існує жодного інструменту, що може використовуватись з метою регулярного аналізу значних об'ємів програмного коду [13], усі протестовані системи розраховані на одиничне використання та не

зберігають жодної інформації для прискорення наступних запусків. Більшість розглянутих систем не може працювати з декількома проектами одночасно, шукаючи дублікати у проектах, що написані на різних мовах програмування. Таким чином, виникає потреба у створенні способу, що дозволяє ефективно визначати дублікати коду за допомогою побудови та аналізу графів та може використовуватись на великих об'ємах вихідного коду, а також може порівнювати код на різних мовах програмування [14].

Для виконання порівняльного аналізу оберемо наступні продукти:

- Duplo(c);
- CCFinder;
- CP-Miner;
- CloneDR.

Кожен з наведених програмних продуктів представляє один рівень порівняння. Усі ці програми задовольняють наступним вимогам:

- можливість роботи з операційними системами Windows та Linux;
- підтримка декількох мов програмування з числа найбільш вживаних;
- наявність інтерфейсу користувача;
- можливість отримання безкоштовної навчальної ліцензії або наявність безкоштовної версії.

Розглянемо детально кожен інструмент, що представляє кожен клас за рівнем порівняння.

Duplo – безкоштовна програма для пошуку дублювання коду з відкритим вихідним кодом. Інтерфейс користувача – текстовий, підтримується режим роботи у командному рядку. Результати порівняння виводяться у вигляді шляхів до файлів, у яких знайдено дублювання коду, з номером конкретного рядка, на якому починається дубльований блок у кожному файлі. Повний текст дубліката виводиться після списку файлів, у яких його було знайдено (рис. 1.10). Офіційно заявлено підтримку

наступних мов програмування:

- C;
- C++;
- Java;
- C#;
- Visual Basic .NET.

Зважаючи на те, що Duplo виконує порівняння на рівні вихідного тексту програми, фактично підтримується значно більше мов програмування. Будь-які дані, представлені у тектовому вигляді, можуть бути проаналізовані так само, як і код на підтримуваних мовах програмування, проте якість результату не гарантується.

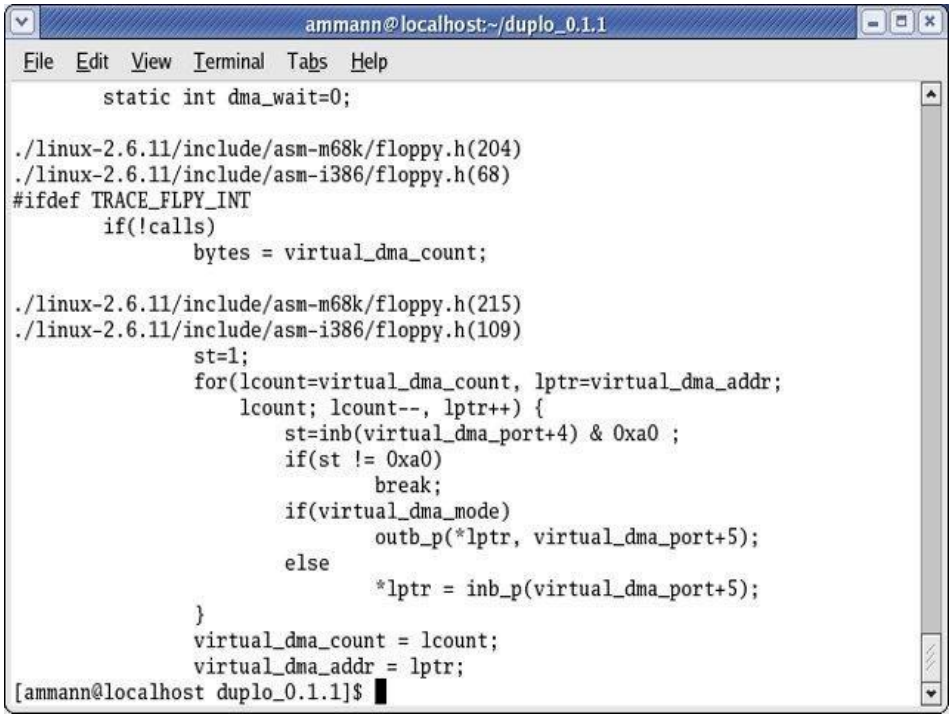


Рисунок 1.10 – Текстовий інтерфейс програми Duplo(C)

Доступні параметри конфігурації, що можуть задаватися за допомогою аргументів командного рядка, можуть містити наступні опції: шлях до файлів або директорій з вихідним кодом для аналізу, включення або виключення певних файлів з аналізу, вивід у документ формату XML та на екран, мінімальна кількість рядків та символів для визначення збігу.

Програма Duplo може визначати лише дублікати першого типу, що підтверджується тестами. Зміна форматування коду, іменування ідентифікаторів, констант призводить до того, що дублікати не можуть бути знайдені. Результати детального огляду програми наведено у таблиці 1.3.

Таблиця 1.3 – Огляд програми Duplo(C)

Назва	Duplo(C)
Клас порівняння	Текст
Рік останнього оновлення	2013
Графічний інтерфейс	Ні
Вивід у файли	Так
Режим командного рядка	Так
Підтримка мов програмування	Java, C#, C, C++, VB.NET, будь-які текстові файли
Розпізнавання дублікатів	Тип 1

Програма CCFinder може визначати дублікати коду першого та другого типу. Тести підтверджують можливість роботи програми при зміні значень ідентифікаторів та форматування коду. Використовуються методи роботи, засновані на наукових роботах дев’яностих та двохтисячних років, тому даний продукт не можна вважати сучасним та ефективним. Його перевагою є наявність повноцінної безкоштовної версії, що не містить жодних обмежень. Вихідний код програми доступний під ліцензією MIT. Серед недоліків можна виділити час останнього оновлення програми, що

датується 2009 роком.

Програма CCFinder має графічний інтерфейс користувача та може виводити результати порівняння у вигляді матриці подібності (рис. 1.11). Програма відповідає своєму класу порівняння та забезпечує ефективний пошук дублікатів коду на мовах програмування Java, C, C#, C++ та інших мовах.

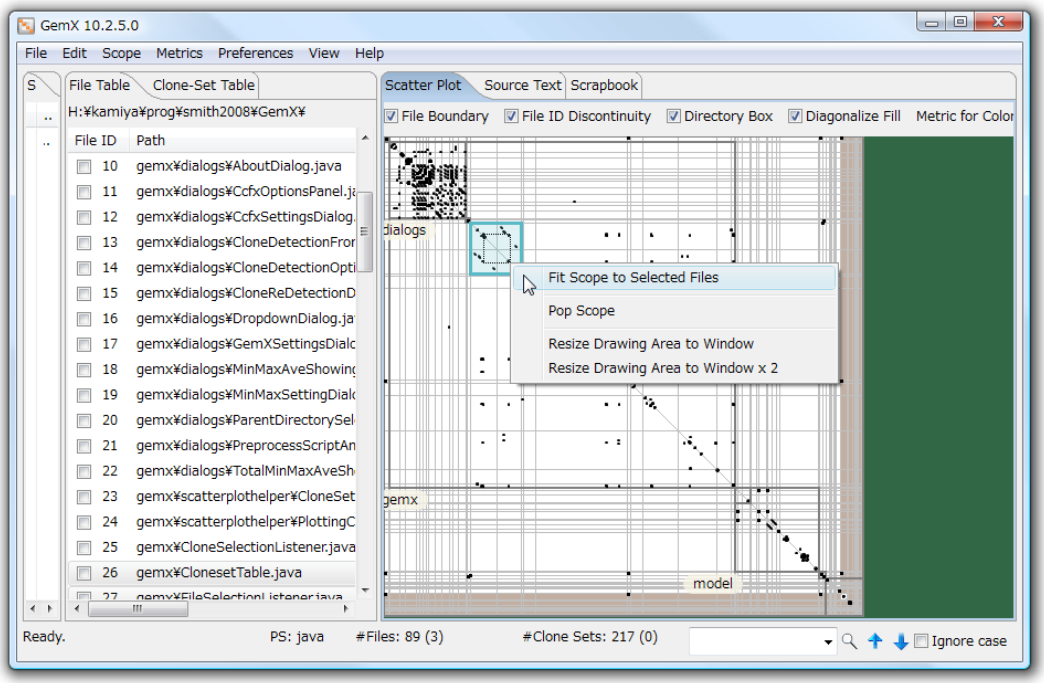


Рисунок 1.11 – Графічний інтерфейс програми CCFinder

Робота з невідомими мовами програмування неможлива, як і з будь-якими текстовими даними. Огляд програми CCFinder наведено у таблиці 1.4.

CP-Miner – це єдина актуальна програма, що використовує підхід на базі порівняння метрик. Програма розроблена та ліцензується компанією Pattern Insight. Це комерційна програма, тому не існує безкоштовної версії. Проте існує можливість надіслати запит на безкоштовну демонстраційну версію.

Програма CP-Miner підтримує мови програмування C та C++, що обумовлено її призначенням – аналізом коду операційних систем Linux та FreeBSD. Використання звичайних підходів ускладнено надзвичайно

великим розміром кодової бази цих систем. Останнє оновлення даного проекту датується 2004 роком, програма не підтримується. Короткий аналіз програми наведено у таблиці 1.5.

Таблиця 1.4 – Огляд програми CCFinder

Назва	CCFinder
Клас порівняння	Лексеми
Рік останнього оновлення	2009
Графічний інтерфейс	Так
Вивід у файли	Так
Режим командного рядка	Так
Підтримка мов програмування	Java, C#, C, C++ та ін.
Розпізнавання дублікатів	Тип 1, тип 2

Сучасною програмою для порівняння синтаксичних дерев з метою пошуку дублювання коду є програма CloneDR, що розроблюється та підтримується компанією Semantic Designs. Метою даної програми є зниження ціни підтримки програмних проектів за рахунок зменшення дублювання коду. Унікальною особливістю даного рішення є можливість автоматичного видалення дублікатів – винесення їх у спільні семантичні блоки. Доступні наступні параметри конфігурації:

- коефіцієнт подібності;
- мінімальний розмір дублікату;
- пошук дублікатів у декларативному коді;

- використання декількох ядер процесора;
- параметри формування звіту.

Таблиця 1.5 – Огляд програми CP-Miner

Назва	CP-Miner
Клас порівняння	Лексеми
Рік останнього оновлення	2004
Графічний інтерфейс	Ні
Вивід у файли	Так
Режим командного рядка	Так
Підтримка мов програмування	C, C++
Розпізнавання дублікатів	Тип 1, тип 2

Існує можливість виведення звітів у вигляді документів формату HTML та XML, проте графічний інтерфейс користувача відсутній. Підтримується наступний набір мов програмування:

- Ada (83, 85);
- COBOL (та його діалекти);
- C/C++;
- C#;
- JavaScript/ECMAScript;
- Java;
- PHP;

- XML та інші мови розмітки.

Так як CloneDR є комерційною програмою, для безкоштовного використання доступна лише обмежена демонстраційна версія. У таблиці 1.6 наведено загальні характеристики розглянутої програми.

Таблиця 1.6 – Огляд програми CloneDR

Назва	CloneDR
Клас порівняння	Абстрактні синтаксичні дерева
Рік останнього оновлення	2018
Графічний інтерфейс	Ні
Вивід у файли	Так
Режим командного рядка	Так
Підтримка мов програмування	Ada, COBOL, Java, C#, C, C++ та ін.
Розпізнавання дублікатів	Тип 1, тип 2, частково тип 3

Серед розглянутих програм немає жодної програми, що виконує порівняння на рівні графів потоку керування через відсутність таких рішень на ринку у даний момент часу.

Порівняння швидкості роботи програм, кількості знайдених дублікатів на синтетичних текстових даних та інші дані наведені у розділі 4 при порівнянні існуючих програм з реалізацією розробленого способу порівняння графів потоку керування.

Розглянувши існуючі рішення можна зробити висновок, що існує необхідність у створенні зручного інструменту для пошуку дублювання

коду, що працює на більш високому рівні за розглянуті рішення та є достатньо ефективним для регулярного застосування. Жодна з розглянутих програм не має оптимізації, що дозволяє пришвидшити подальші запуски аналізу.

2. СПОСОБИ ЗБЕРІГАННЯ, ПОБУДОВИ ТА ПОРІВНЯННЯ ПОДІБНОСТІ ГРАФІВ ПОТОКУ КЕРУВАННЯ

Однією з основних вимог до системи, що розробляється, є ефективність у роботі з великими за об'ємом проектами. Кількість етапів аналізу програми, що виконуються до безпосереднього порівняння та вимагають обходів файлів або дерев, має бути мінімальною. Порівняння програм на рівні графів потоку керування має переваги у швидкості побудови, тому рівень графів потоку керування застосовується як кінцевий етап трансформації вхідного коду.

За вимогами до системи вона повинна мати можливість порівняння проектів, що використовують різні мови програмування. Таким чином виникає необхідність зведення програм до вигляду, у якому відмінності мови програмування не впливають на можливість пошуку дублікатів. Представлення програми у вигляді графу потоку керування - перший крок до незалежності від мови програмування, так як графи потоку керування для однакових програм, що написані на різних мовах програмування (за умови існування відповідних циклічних та умовних конструкцій у обох мовах), будуть мати однакову форму. Проте зміст кожного вузла залежить від мови програмування та має бути приведений до узагальненого вигляду у майбутньому.

Для відповідності обом вимогам обрано наступну послідовність основних трансформацій:

- лексичний аналіз;
- синтаксичний аналіз;
- побудова графу потоку керування.

2.1. Використання графу потоку керування для порівняння програмного коду

Граф потоку керування будується сучасними компіляторами та аналізаторами коду після виконання лексичного та синтаксичного аналізу. Кожна вершина графа потоку керування програми містить одну інструкцію програми, або послідовність інструкцій, що не містить інструкцій умовної передачі керування та виконується послідовно.

Побудова графа потоку керування виконується за допомогою обходу синтаксичного дерева в глибину до рівня окремих інструкцій. Послідовні інструкції додаються до поточної вершини, а інструкції, що мають декілька можливих шляхів передачі керування - умовні оператори (рис. 2.1), оператори безумовного переходу, оператори виключення (рис. 2.2) та оператори циклів (рис. 2.3) - завершують поточну вершину та створюють інші, що відповідають можливим шляхам виконання з даної інструкції.

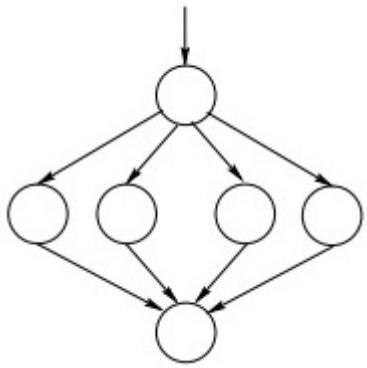


Рисунок 2.1 – Граф потоку керування для умовної інструкції типу switch

Граф потоку керування містить інформацію про те, як виконується програма. Усі шляхи, що існують у графі потоку керування, відповідають існуючим шляхам виконання програми. Граф потоку керування не містить жодної інформації про форматування коду та про наявність і зміст коментарів у вихідних файлах [5]. Таким чином можна зробити висновок, що дублікати першого типу, які відрізняються лише форматуванням,

будуть гарантовано розпізнаватися будь-яким способом пошуку дублікатів, крім текстового.

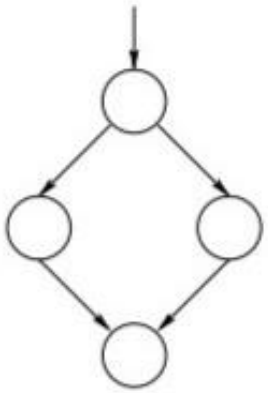


Рисунок 2.2 – Граф потоку керування для умовної інструкції типу if з додатковою гілкою else

Використання графу потоку керування завжди гарантує розпізнавання дублікатів першого типу.

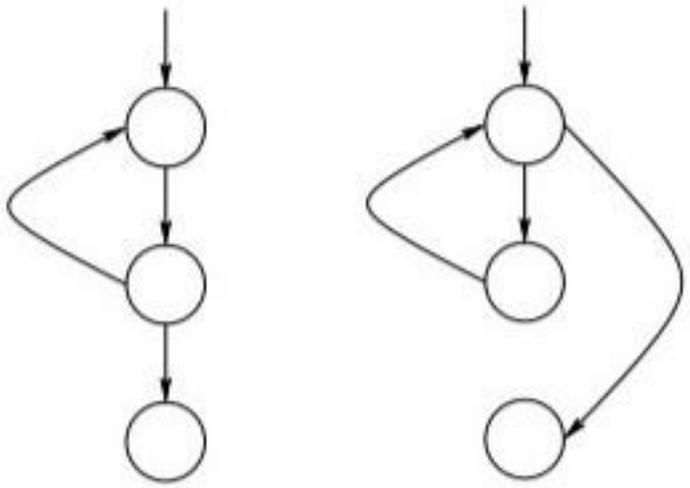


Рисунок 2.3 – Графи потоку керування для циклів until та while відповідно

Дублікати другого типу можуть містити коментарі, значення змінних та констант можуть відрізнятись, а також вирази можуть містити певні відмінності. Можливість розпізнавання дублікатів другого типу залежить від конкретної реалізації та не гарантується способом розпізнавання.

У способах пошуку дублікатів, що виконують порівняння на рівні

графів, принципова можливість та якість розпізнавання дублікатів другого типу залежить від реалізації функції порівняння вершин графа. Найчастіше використовуються наступні підходи:

- посимвольне порівняння;
- порівняння лексем;
- порівняння синтаксичних піддерев;
- порівняння лексем з ігноруванням значень констант та ідентифікаторів;
- порівняння синтаксичних піддерев з ігноруванням певних елементів;
- порівняння результатів хеш-функції.

Посимвольне порівняння виконується лише за умови доступного текстового представлення синтаксичного піддерева інструкції, так як конвертація даних у текст для кожної вершини програми – довгий та складний процес. Порівняння лексем відрізняється лише тим, що необхідно мати доступне лексичне представлення синтаксичного піддерева.

Порівняння синтаксичних піддерев виконується частіше, так як графи, зокрема граф потоку керування, будуються на основі синтаксичного дерева. Таким чином синтаксичні дерева кожної інструкції гарантовано доступні, якщо синтаксичний аналіз було виконано у повному об'ємі. Порівнянням синтаксичних піддерев є процес співставлення кожної вершини одного піддерева відповідній вершині іншого піддерева. При будь-яких відмінностях вузли графу потоку керування (або іншого графу) вважаються різними.

Вищеописані способи повного порівняння не можуть використовуватись для пошуку дублікатів другого типу, тому що вимагають повного збігу. Тому з метою підвищення якості пошуку дублікатів використовуються способи з ігноруванням деяких відмінностей. При порівнянні на рівні лексем та синтаксичних піддерев, існує

можливість ігнорувати конкретні значення констант та ідентифікаторів, тому при їх використанні у функції порівняння вузлів різні ідентифікатори та числові або текстові константи будуть визначатись як однакові. Проте інформація про тип константи зберігається, тому текстові константи не можуть дорівнювати числовим (за виключенням мов програмування, що не мають чіткого розділення на числові та нечислові типи).

Порівняння вузлів на рівні синтаксичних піддерев дозволяє виключити з порівняння наступні елементи:

- вирази будь-якої складності;
- ідентифікатори;
- фактичні аргументи функцій;
- блоки ініціалізації;
- параметри умовних операторів;
- константи.

Ігнорування вищезазначених конструкцій при порівнянні вузлів графа потоку керування гарантує знаходження усіх дублікатів другого типу, але призводить до помилкового визначення дублікатами інших даних, що не є дублікатами. Для боротьби з даною проблемою можливо частково вимикати ігнорування або використовувати евристичні методи, що будуть визначати підозрілість кожного конкретного фрагменту коду за більшою кількістю параметрів, включаючи розмір фрагменту та кількість повних збігів.

Окремими способом порівняння вузлів є порівняння результатів застосування хеш-функції для кожного з них (рис. 2.4). Хеш-функція може розраховуватись з можливістю ігнорування усіх вищезазначених синтаксичних елементів, або частково зберігати інформацію про їх зміст для окремого визначення повного та неповного збігу. Хеш-функції можуть розраховуватись за текстовим представленням вузла, за лексичним представленням вузла, за допомогою розрахунку хеш-функції від хеш-функцій усіх піддерев синтаксичного дерева.

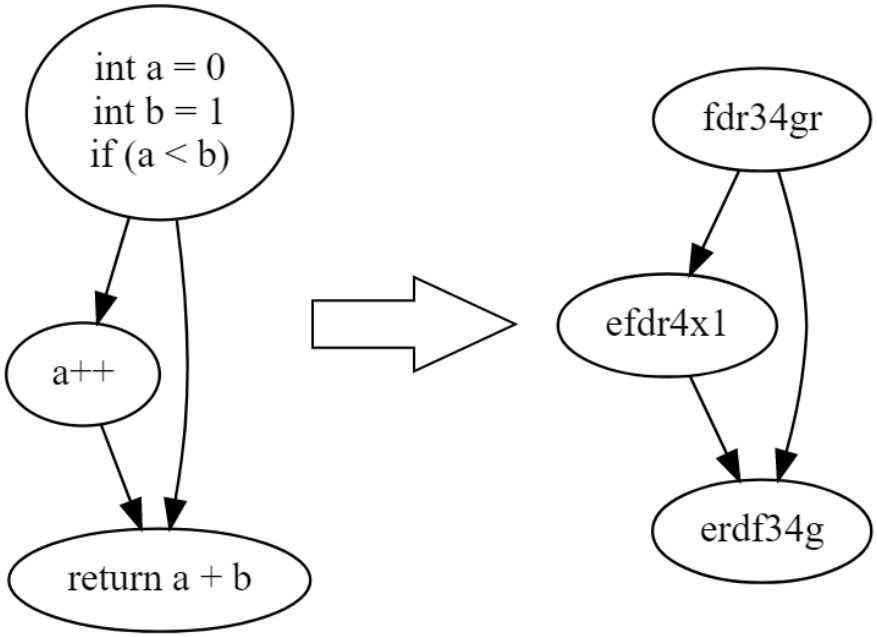


Рисунок 2.4 – Конвертація вузлів графу за допомогою хеш-функції

Для визначення дублікатів третього типу необхідно мати можливість визначення неповних збігів. Якщо кожен вузол графа містить лише одну інструкцію (за варіантом побудови графа потоку керування), то задача визначення доданих та змінених інструкцій виконується на рівні порівняння графу. Проте за типовим способом побудови графів потоку керування кожен вузол може містити будь-яку кількість інструкцій, що завжди виконуються послідовно, тому використовуються вищеназвані способи визначення неповного збігу для кожного вузла графа.

Представлений спосіб використовує графи потоку керування, побудовані за особливими правилами, що покращують майбутню конвертацію у текстовий вигляд.

Розроблений спосіб базується на виконанні незворотного перетворення вершин за допомогою часткового семантичного аналізу змісту інструкцій.

2.2. Задача визначення подібності графів

Задача визначення подібності графів – це задача визначення ізоморфізму графів потоку керування. В області пошуку дублювання програмного коду зміст графу потоку керування має не менш важливе значення, ніж його форма, тому перш за все необхідно визначити спосіб порівняння вершин графа.

Для порівняння графів найзручніше використовувати інваріантне представлення графу. Представлений спосіб порівняння графів базується на представленні графів у вигляді пар вершин, з'єднаних ребром. Кількість записів відповідає кількості ребер графа. Кожна вершина записується у вигляді набору символів.

Вершина графу потоку керування може містити один або декілька наборів символів, кожен з яких відповідає одній програмній інструкції. Текст програмного коду не зберігається, завдяки чому досягається незалежність від форматування коду, мови програмування, коментарів, іменування ідентифікаторів та значень констант.

Кожен набір символів має включати наступну інформацію:

- тип інструкції;
- конкретний граматичний сорт інструкції;
- формальні параметри та аргументи, або ознака їх відсутності у кожній інструкції;
- інформація про типи даних, що використовуються у даній інструкції.

Тип інструкції визначає, чи є конкретна інструкція інструкцією передачі керування, інструкцією присвоювання, викликом функції тощо. Кожна мова програмування має свій набір інструкцій та їх можливих типів, тому було визначено найбільш універсальні типи, що можуть використовуватись у більшості імперативних мов, а також у деяких функціональних мовах програмування (таблиця 2.1).

Конкретний граматичний сорт дозволяє розрізняти такі елементи, як різні типи циклів, різні типи викликів та клас спеціальних елементів. Завдяки використанню другого символу замість розширення можливих значень першого символу досягається можливість визначення часткового збігу вузлів графу у випадках, коли код було частково змінено, наприклад, при зміні типу циклічної конструкції.

Таблиця 2.1 – Класифікація типів інструкцій

Тип	Приклади відповідних інструкцій

Циклічний оператор	while (true) {};
	foreach(var x in y) {};
	for (;;) {};

Умовний оператор	if (true) {}; let x = if thue then y else z; switch (x) {};
Оператор повернення	return;

Оператор декларування або присвоювання	int x = y;
	x = 10 + 1;
	string s1, s2;
	double pi = Math.Round(3.14);

Виклик функції (метода)	Call(); System.WriteLine(x);
Спеціальний елемент	Початок та закінчення метода, некласифікований елемент

Формальні параметри чи аргументи визначають наявність або відсутність у функції циклічної (рис. 2.5) чи умовної конструкції (рис. 2.6),

операторі присвоювання та повернення аргументів. Для функцій або методів, що приймають декілька аргументів, кожен аргумент оцінюється та зберігається інформація лише про найважливіший елемент. Сортування елементів за важливістю виконується в порядку, показаному в таблиці 2.2.

Таблиця 2.2 – Класифікація типів аргументів

Тип	Приклади відповідних аргументів
Вираз	x + b * 2
Локальна змінна	Ідентифікатор, що декларується у області видимості даної функції
Поле класу	Ідентифікатор, що не декларується у області видимості даної функції
Виклик функції	Math.Round(3.14)
Аргумент функції	Змінна, що використовується у списку формальних аргументів функції
Константа	100500.0f
Об’єкт	new Object()
Невизначено	Використовується у випадках, коли неможливо визначити тип

Інформація про тип об’єкту потребує більш глибокого семантичного аналізу програми. Будь-які сучасні мови програмування мають можливість описувати нові структури даних та використовувати їх у якості типів, тому неможливо розрізняти кожен фактичний тип даних при їх проекції на кінцеву множину можливих значень символів. Для вирішення даної проблеми вводиться класифікація, що поділяє можливі типи даних на наступні класи:

- рядок;

- число;
- послідовність;
- тип, створений користувачем;
- невідомий тип.

Рядок – тип, що існує у більшості сучасних мов програмування та представляє текстові дані. У мовах програмування Java та C# використовується тип «String».

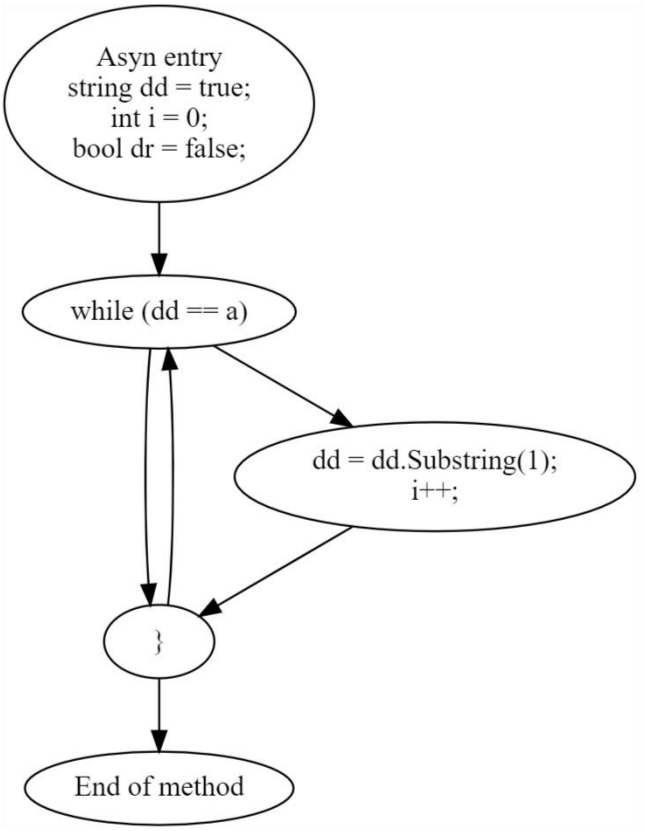


Рисунок 2.5 – Модифікований граф потоку керування для циклічної інструкції

Числові типи з плаваючою або фіксованою крапкою присутні у більшості мов програмування з сильною типізацією, але не всі мови розрізняють типи різної розрядності, тому числові типи було узагальнено для можливості порівняння програм на різних мовах програмування.

Послідовність елементів – тип даних, що представлений у вигляді масивів або списків у кожній сучасній мові програмування. Так як для

кожної мови існує свій набір перелічуваних типів, вони були узагальнені у один тип для спрощення процесу порівняння. Зміна одного типу, що представляє послідовність, на інший, не вплине на якість розпізнавання дублікатів коду.

Типи, що створені користувачем – це типи, що не входять до стандартних типів мови програмування. Різні мови програмування дозволяють створення класів, структур, записів та перелічуваних типів. Назва типу та його вміст залежить лише від програміста, тому усі подібні типи класифікуються однаково.

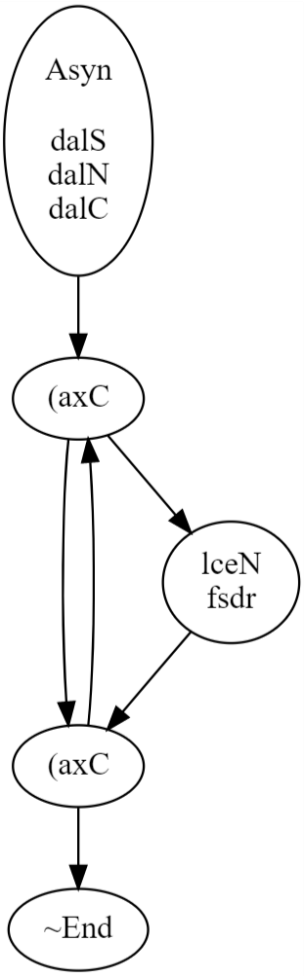


Рисунок 2.6 – Стиснене представлення графа потоку керування для циклічної інструкції

Визначення типу даних є непростою задачею. Для тривіальних випадків, коли тип даних вказано безпосередньо у декларації функції,

змінної або він може бути виведений з типу константи, визначення типу даних не потребує глибокого семантичного аналізу. У тих випадках, коли тип задається поза межами функції, що розглядається, або тип є динамічним – використовується клас «невідомий тип». Виконання аналізу для визначення кожного типу дозволяє підвищити точність, але значно погіршує швидкість виконання порівняння, тому аналіз типів не виконується.

У розробленому способі використовується 4 символи, що відповідають чотирьом розглянутим ознакам. Збільшення кількості символів незначно покращує якість розпізнавання та зменшує кількість помилкових збігів, проте вимагає значно більше часу на їх розрахунок. Зменшення кількості символів призводить до різкого зростання числа помилкових збігів, але дозволяє незначно підвищити продуктивність.

2.3. Спосіб порівняння для визначення відповідності графів потоку керування

Для пошуку дублікатів коду між кожною існуючою парою функцій або методів необхідно виконати порівняння відповідних графів потоку керування. Для порівняння даних необхідно подання графів у вигляді, у якому спрощується процес визначення ізоморфізму графів. Представлений спосіб порівняння базується на порівнянні графів у текстовому представленні у вигляді послідовності їх ребер. Такий формат запису забезпечує можливість порівняння графів незалежно від їх форми (рис. 2.7).

Рисунок 2.7 – Ізоморфні графи

Процес порівняння поділяється на 3 рівні, на кожному з яких визначається подібність:

- граф потоку керування;
- ребро графа потоку керування;
- вершина графа потоку керування (рис. 2.8).

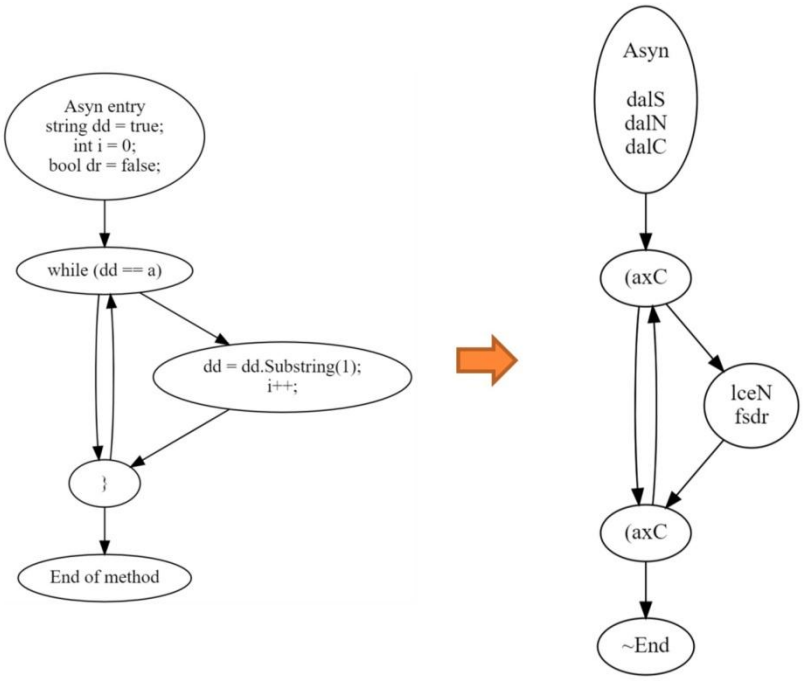


Рисунок 2.8 – Стиснення графа

Граф потоку керування, що використовується у порівнянні – текстовий файл, що відповідає конкретній функції або методу (рис. 2.9). Крім інформації про форму та зміст графа необхідно також зберігати інформацію про шлях до файлу вихідного коду, назву файлу та семантичної одиниці, назву функції та позицію у файлі (номер рядку,

номер позиції). У визначенні подібності ця інформація участі не приймає, але використовується для виведення конкретних фрагментів коду, що дублюється.

Графи позначаються підозрілими у випадку, якщо їх коефіцієнт подібності перевищує певне порогове значення. Це значення у більшості існуючих методів лежить у проміжку від 0.7 до 0.9. За замовчуванням рекомендується встановлювати значення рівним 0.7, але якщо важливо мінімізувати кількість неточних збігів – користувач має можливість збільшити порогове значення. Таким чином забезпечується універсальність способу порівняння та можливість конфігурації в залежності від задач, що вирішуються.

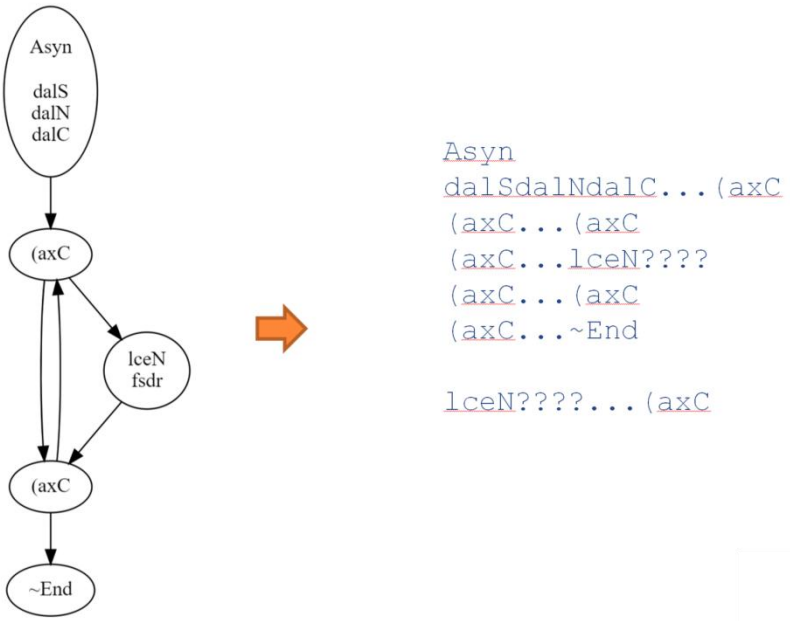


Рисунок 2.9 – Приклад представлення графа у текстовому вигляді за допомогою послідовності ребер

Відповідність графів розраховується за формулами (1) та (2).

$$f_1 = \frac{|nodes(X_1) \cap nodes(X_2)|}{|nodes(X_1) \cup nodes(X_2)|}, \quad (1)$$

$$f_2 = \frac{|nodes(X_1) \cap nodes(X_2)|}{|nodes(X_1)|}, \quad (2)$$

де функція f_2 визначає кількість подібних інструкцій у фрагментах x_1 та x_2,

функція f_2 визначає оригінальність фрагменту x_1 відносно фрагменту x_2, $nodes(x)$ – множина ребер графа потоку керування.

Перша формула відповідає загальному коефіцієнту подібності. Цей показник зручно використовувати при виконанні статистичного аналізу, він дозволяє оцінити загальний стан вихідного коду продукту та прийняти рішення про необхідність виконання рефакторингу.

Друга формула може використовуватись з метою оцінки унікальності, порівняння декількох кодових баз. Значення другого індексу подібності залежить від напряму порівняння.

Мінімальний розмір графа, при якому є сенс включати метод або функцію до множини порівняння, визначається розміром його текстового представлення. Розмір представлення графа для функції без розгалуження можна визначити за наступною формулою (3):

$$S = 4x + 4, \tag{3}$$

де x відповідає кількості послідовних інструкцій функції. Якщо прийняти, що дублікатом може бути функція за наявності у ній чотирьох чи більше інструкцій (без врахування розгалуження), то можна виключити з розгляду усі графи потоку керування, розмір текстового представлення яких, за умови відповідності одного записаного символу одному байту пам’яті, менше або дорівнює 20 байт. Цей розмір не включає додаткову інформацію, таку як шлях до файлу та позиція у файлі.

Якщо функція містить інструкції умовного переходу або цикли, мінімальний запис такої функції буде містити більше ніж одне ребро та вийде за обмеження в 20 байт вже за наявності двох інструкцій, одна з яких – інструкція умовної передачі керування.

Визначення відповідності ребер графа – це окрема задача, що вирішується на рівні порівняння ребер. Так як кожне ребро графа описується у вигляді пари вершин, які воно з’єднує, відповідність двох ребер повністю визначається відповідністю їх початкових та кінцевих вершин. Відповідність ребер розраховується за формулою (4):

$$v(x) = \begin{cases} 1, i(x) \geq k \\ 0, i(x) < k \end{cases} \quad (4)$$

Коефіцієнт k задається користувачем та перевищує або дорівнює пороговому значенню подібності. За замовчуванням рекомендується значення 0.8. Зменшення коефіцієнту k дозволяє знаходити більше дублікатів третього типу, але значно збільшує кількість помилкових збігів на невеликих функціях.

Для порівняння інструкцій, представлених у вигляді послідовності символів, розраховується індекс подібності за наступною формулою (5):

$$i = \frac{2S}{(2S+L+R)} \quad (5)$$

У наведеній формулі S – кількість однакових символів, L та R – кількість унікальних символів у першому та другому елементі відповідно.

3. СТРУКТУРА СИСТЕМИ ДЛЯ ПОРІВНЯННЯ ГРАФІВ ПОТОКУ КЕРУВАННЯ ТА ЇЇ ТЕСТУВАННЯ

3.1. Граматика мови C#

Для створення системи пошуку дублікатів програмного коду на основі порівняння дерев потоку керування, було обрано мову програмування C#. Мова C# є відомою об'єктно-орієнтованою мовою програмування загального призначення, що широко використовується у комерційній розробці програмного забезпечення. Вибір мови C# обумовлено наступними умовами:

- висока популярність мови C# у бізнесі;
- наявність програмної платформи та компілятора Roslyn з відкритим вихідним кодом;
- наявність широкої підтримки з боку компанії Microsoft та спільноти розробників;
- існуючі проекти з відкритим вихідним кодом, що мають достатньо великий об'єм коду для тестування.

Мова програмування C# створена для написання зручного та ефективного коду, що може використовуватись у великих проектах. Сучасні нововведення мови C# спрямовані на зменшення кількості коду, необхідного для реалізації алгоритмів, та додавання можливостей функціонального програмування. Синтаксис мови активно змінюється, тому для підтримки найсучасніших стандартів рекомендується використовувати офіційні інструменти від компанії Microsoft, замість створення та підтримки власних рішень. Актуальний лексичний та синтаксичний аналізатор, компілятор та семантичний аналізатор, входять до складу проекту Roslyn. Розроблена система використовує Roslyn у якості парсера та працює з вихідним кодом мови C# на рівні абстрактного синтаксичного дерева.

Основними синтаксичними одиницями, що використовуються у побудові графів потоку керування, є програмні інструкції, що відповідають нетерміналу «statement» у граматиці мови C# (рис. 3.1).

```
statement
        : (labeled_statement) => labeled_statement
        | (declaration_statement) => declaration_statement
        | embedded_statement
        ;
embedded_statement
        : block
        | empty_statement
        | expression_statement
        | selection_statement
        | iteration_statement
        | jump_statement
        | try_statement
        | checked_statement
        | unchecked_statement
        | lock_statement
        | using_statement
        | yield_statement
        | embedded_statement_unsafe
        ;
```

Рисунок 3.1 – Частина граматики мови C#, що відповідає програмним інструкціям

Для побудови графу потоку керування та необхідно виконати обхід синтаксичного дерева в глибину.

3.2. Побудова графа потоку керування

Граф потоку керування будується після виконання лексичного та синтаксичного аналізу вихідного коду. Для побудови міжпроцедурного графу потоку керування необхідно також побудувати граф викликів програми. Так як міжпроцедурний граф потоку керування є досить складним у побудові та його складність зростає з кожним викликом

функції, повний міжпроцедурний граф потоку керування не будується (рис. 3.2). Побудова графу потоку керування для кожного окремого методу чи функції є не лише ефективним, а й більш точним методом з точки зору пошуку дублікатів. Абсолютна більшість дублікатів - це дублікати функцій, методів чи їх відносно незалежних частин, тому побудова часткових графів для кожного методу значно пришвидшує процес порівняння графів. Недоліком такого способу є те, що втрачається можливість пошуку дублікатів, якщо один з них було розбито на невеликі за обсягом окремі функції, але їх відсоток досить малий у реальних проектах.

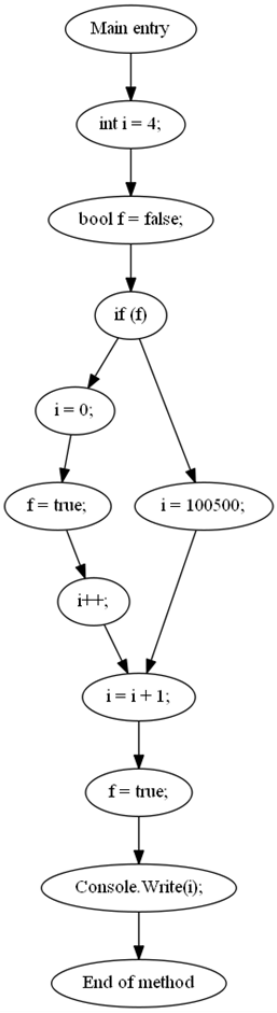


Рисунок 3.2 – Граф потоку керування без модифікацій

Для побудови графу потоку керування виконується наступна послідовність дій:

- обхід абстрактного синтаксичного дерева для пошуку методів у кожному класі;
- рекурсивний обхід абстрактного синтаксичного дерева кожного методу для пошуку інструкцій, включаючи інструкції умовної передачі керування, а також вкладені блоки інструкцій;
- обхід побудованого графу потоку керування в глибину та об'єднання елементів, що відповідають критеріям об'єднання (рис. 3.3).

Обхід виконується за наступним алгоритмом. Для кожного синтаксичного дерева, що відповідає файлу вихідного коду, визначаються усі методи кожного класу, описаного у файлі. Для процедурних мов - усі функції та процедури. Наступним кроком, що виконується для кожного метода, є обхід дерева для метода з визначенням програмних інструкцій. Для кожної інструкції, що змінює потік виконання програми, створюються відповідні зв'язки у графі потоку керування, а також виконується обхід її внутрішніх блоків. Для звичайних інструкцій створюються зв'язки з наступною та попередньою інструкцією. Першим вузлом графу завжди є вузол початку графу, що не відповідає жодній програмній інструкції та має спільне ребро з першою інструкцією. Останнім вузлом графу є кінцевий вузол, що має зв'язки з останньою інструкцією та кожною інструкцією повернення з метода чи функції.

Метою об'єднання деяких вузлів графу є зменшення кількості інструкцій для подальшої обробки та конвертації. У текстовому представленні граф записується у вигляді послідовності вершин, з'єднаних ребром, тому зменшення кількості таких вершин дозволяє виконувати порівняння більш ефективно. До того ж, завдяки визначенню неповних збігів елементів та групуванню послідовних вузлів, існує можливість

визначати дублікати третього типу.

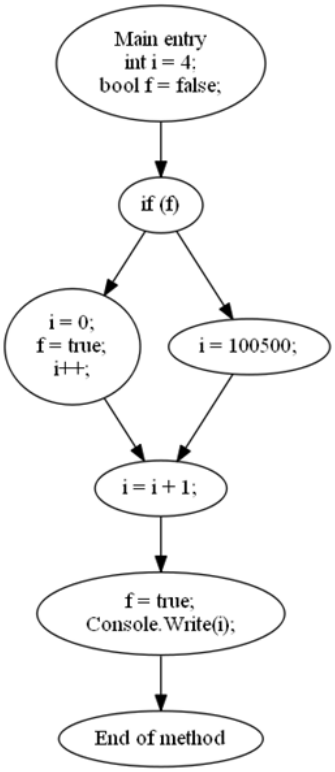


Рисунок 3.3 – Модифікований граф потоку керування після процедури об'єднання

Об'єднання виконується, якщо елементи, що мають бути об'єднані, відповідають наступним критеріям:

- останній елемент містить лише одне вихідне ребро;
- кожен елемент містить лише одне вхідне ребро.

Подальші трансформації графу потоку керування, такі як нормалізація та оптимізація циклів, що можуть виконуватись з метою підвищення якості розпізнавання дублікатів третього типу, не використовуються у розробленому способі через те, що вимагають багато часу, але можуть використовуватись для підвищення точності ціною ефективності.

3.3. Перетворення графа потоку керування та представлення графа у стисненому вигляді

Перетворення графу потоку керування виконується з метою стиснення інформації про зміст вершин графу. Вершини графу потоку керування у звичайному вигляді містять синтаксичні дерева або блоки синтаксичних дерев. Кожне синтаксичне дерево представляє частину граматики мови програмування (у контексті розробленої програми - це мова C#), що відповідає одній програмній інструкції. Інформація, що міститься у повному синтаксичному дереві, є надлишковою для задач пошуку дублювання коду. Проте виключення цієї інформації повністю - видалення синтаксичного дерева – також не може бути вирішенням проблеми через те, що залишається інформація лише про форму графа, а не про його зміст. Цієї інформації недостатньо для виконання пошуку дублікатів коду.

Оптимальним з точки зору витрат часу, об'єму інформації та якості порівняння є варіант, що включає трансформацію представлення інформації у вигляді синтаксичних дерев у представлення інформації на базі певної кількості ознак, що можуть приймати обмежену множину значень.

У розробленій тестовій програмі, що виконує пошук дублікатів у програмних проектах на мові C#, реалізовано аналіз чотирьох ознак, що відповідають наступним характеристикам:

- тип синтаксичної інструкції;
- підтип синтаксичної інструкції;
- тип аргументу (або аргументів);
- тип даних, що використовується у інструкції або повертається з функції.

Конвертація виконується за наступним алгоритмом:

- обхід кожної вершини графа потоку керування;

- послідовний обхід кожного синтаксичного дерева у вершині графа потоку керування;
- визначення типу синтаксичної інструкції, що описаний кореневим елементом синтаксичного дерева;
- визначення підтипу синтаксичної інструкції, що залежить від кореневого елемента або інших елементів синтаксичного дерева;
- пошук аргументів програмної інструкції;
- якщо аргумент знайдено, визначення його типу, типу даних;
- якщо знайдено більше одного аргументу - виконується сортування аргументів за рівнем їх важливості;
- визначається тип даних за допомогою опису типів, присутніх у синтаксичному дереві;
- якщо тип не знайдено – використовується таблиця типів для визначення типу даних;
- якщо тип не знайдено у таблиці – записується ознака невідомого типу;
- конкатенація результатів конвертації кожного синтаксичного дерева у вершині графа потоку керування;
- створення нового графу потоку керування, що містить модифіковані вершини. Зв'язки залишаються незмінними.

Для визначення типу синтаксичної інструкції виконується два кроки аналізу. Якщо кореневий елемент абстрактного синтаксичного дерева відповідає певній граматичній одиниці, що міститься у таблиці відомих інструкцій - виконується пряме перетворення відповідно до таблиці 3.1. У іншому випадку, коли інструкція не входить до множини відомих інструкцій, виконується обхід синтаксичного дерева з метою пошуку вкладених інструкцій. Якщо такі інструкції будуть знайдені – виконується визначення типу інструкції на основі її вкладених інструкцій. Якщо інструкція не містить внутрішніх інструкцій та її тип не визначено –

записується ознака невідомої інструкції. Подальші кроки для даної інструкції не виконуються. Так як множина інструкцій мови програмування C# обмежена, то можливо реалізувати табличне перетворення для кожної можливої інструкції, але використання більш глибокого аналізу робить представлений спосіб більш універсальним та нечутливим до можливих змін граматики (якщо такі зміни не є критичними).

Таблиця 3.1 – Приклад таблиці співвідношення типів синтаксичних інструкцій

Тип інструкції	Тип граматичної одиниці
Умовна інструкція	if-statement, switch-statement
Циклічна інструкція	for-statement, while-statement, foreach-statement, do-statement
Інструкція переходу	continue-statement, break-statement;
Інструкція виклику	Call-statement
Інструкція присвоювання	Expression-statement; initialization-statement

Визначення підтипу синтаксичної інструкції повністю залежить від визначеного типу інструкції. В загальному випадку визначаються наступні умови:

- тип циклічної чи умовної конструкції;
- вид функції, що викликається (локальна, системна, інша);
- ініціалізація або декларування змінної;
- присвоєння або зміна значення змінної.

Завдяки використанню двох символів для кодування ознак розроблений спосіб дозволяє покращити розпізнавання дублікатів коду

третього типу.

Визначення наявності та типу аргументів виконується за допомогою обходу синтаксичного дерева. Конкретні елементи, що використовуються у якості аргументів, визначаються типом та підтипом конкретної синтаксичної інструкції. У таблиці 3.2 наведено відповідність аргументів та типів синтаксичних конструкцій, що визначаються у мові програмування C#. Для інших мов програмування таблиця відповідності може відрізнятись.

Таблиця 3.2 – Аргументи програмних інструкцій у мові C#

Тип	Аргумент	Декілька аргументів
Циклічний оператор	Умова	Ні
Умовний оператор	Умова	Ні
Оператор повернення	Вираз, що повертається	Ні
Оператор декларування або присвоювання	Значення, що присвоюється, або задане початкове значення	Ні
Виклик функції (метода)	Аргумент функції (метода)	Так
Спеціальний елемент	Не має аргументів	Ні

Аргументи класифікуються за типом аргументу. Розрізняють наступні типи:

- вираз;
- локальна змінна;
- поле класу;
- виклик функції;
- аргумент функції;
- константа;

- об’єкт.

Для визначення типу аргументу використовується таблиця ідентифікаторів. Таблиця ідентифікаторів будується під час обходу графа потоку керування в ширину та містить інформацію про тип даних (якщо відомо), походження ідентифікатора та місце його декларування (для того, щоб коректно оброблювати випадки перекриття ідентифікаторів).

```
let convertAtom (c: Context) (s: SemanticModel) (e: ExpressionSyntax) = // Context -> SemanticModel -> ExpressionSyntax -> Atom
    if (e = null) then
        if ParseUtils.DEBUG_MODE then
            printfn "convertAtom: no expression received"
        // default value
        {DataType = UndefinedType; Kind = Undefined; Text = "---"}
    else
        let t = s.GetTypeInfo(e)

        let typeinfo = typeInfoToTypeClass t

        let aKind =
            match e with
            | :? LiteralExpressionSyntax as lit -> Literal
            | :? IdentifierNameSyntax as id ->
                let sym = s.GetSymbolInfo(id).Symbol
                if sym = null then
                    if ParseUtils.DEBUG_MODE then printfn "convertAtom: sym is null in %A" <| id.ToString()
                    Undefined
                else
                    match sym.Kind with
                    | SymbolKind.Local -> LocalVar
                    | SymbolKind.Field | SymbolKind.Property -> ClassField
                    | SymbolKind.Parameter -> Argument
                    | _ -> Expression
            | :? InvocationExpressionSyntax -> MethodCall
            | :? MemberAccessExpressionSyntax -> ExternalField
            | :? ElementAccessExpressionSyntax ->
                LocalVar
            | :? ObjectCreationExpressionSyntax -> ObjConstruction
            | _ -> Expression

        {DataType = typeinfo; Kind = aKind; Text = e.ToString()}
```

Рисунок 3.3 – Алгоритм конвертації аргументів мови C# в узагальнений тип аргументів

Походження ідентифікатору може бути одним з наступних варіантів:

- локальна змінна;
- аргумент функції;
- спеціальний ідентифікатор.

Локальна змінна визначається за допомогою пошуку назви змінної у таблиці ідентифікаторів. Якщо ідентифікатор існує у таблиці на момент обходу – він є доступним у даній точці програми та відповідає локальній змінній.

Якщо ідентифікатор відсутній у даній змінній, але звертання до нього відбувається так само, або за допомогою кваліфікатора «this» – можна зробити припущення, що аргумент є полем даного класу.

Виклик функції визначається за наявністю синтаксичної конструкції «виклик функції» на верхньому рівні синтаксичного дерева аргументу, що розглядається.

Якщо ідентифікатор присутній у таблиці ідентифікаторів та має походження «аргумент функції» – він однозначно класифікується як аргумент функції.

Константа визначається у випадку, якщо на верхньому рівні синтаксичного дерева аргументу, що розглядається, знаходиться нетермінал, що відповідає константі. Для мови програмування C# константи відповідають синтаксичному типу «literal» та його підтипам (в залежності від типу даних).

Якщо аргумент є виразом створення об'єкта – визначається тип аргументу «об'єкт». У мові програмування C# вираз створення об'єкта позначається ключовим словом «new».

У всіх інших випадках визначається тип аргументу «вираз».

Визначення типу даних виконується декількома способами, в залежності від типу синтаксичної конструкції, що розглядається:

- семантичний аналіз типів, що використовуються явно у даній конструкції;
- пошук ідентифікаторів, що використовується у синтаксичній конструкції, у таблиці відомих типів;
- визначення типу константи;
- виведення типів на основі виразів.

Найпростішим елементом для виконання аналізу типів є інструкція декларування змінної, тип якої вказано явно. На основі аналізу інструкцій декларування будується таблиця відомих типів, що містить співвідношення між ідентифікаторами та їх типом (табл. 3.3). Сигнатура

функції також використовується як джерело даних для заповнення таблиці відомих ідентифікаторів. Додавання аргументів функції до таблиці відомих типів виконується під час обходу вхідної вершини графа потоку керування, тому під час обходу усіх інших синтаксичних інструкцій у інших вершинах графа, інформація про ідентифікатори параметрів функції та їх типи є доступною.

```
let typeInfoToTypeClass (t: TypeInfo) = // TypeInfo -> TypeClass
    let realType = match t.Type with
        | null -> t.ConvertedType
        | _ -> t.Type
    if realType = null then ClassType else //just class type if type is unknown
        if (isNumericTypeName realType.Name) then Numeric else
            match realType.TypeKind with
            | TypeKind.Array -> Array
            | TypeKind.Class -> if realType.Name = "String" then String else ClassType
            | TypeKind.Delegate -> ClassType
            | TypeKind.Interface -> ClassType
            | TypeKind.Struct -> ClassType
            | _ -> ClassType
```

Рисунок 3.4 – Алгоритм конвертації типу даних мови C# в узагальнений тип даних

У мові C# активно використовується автоматичне виведення типів компілятором за допомогою ключового слова «var», що використовується замість явного вказання типу. Використання даного ключового слова дозволяє зробити програмування зручнішим, але робить семантичний аналіз типів у мові програмування C# складнішим. У таких випадках для визначення фактичного типу змінної використовується аналіз констант або виразів. Виведений тип записується до таблиці відомих типів. Якщо тип не може бути виведено – запис у таблицю відомих типів відбувається з позначенням невідомого типу.

Для інструкцій, що не містять явно вказаного типу даних, але передбачають певну роботу з даними, визначається тип даних їх аргументів. Якщо інструкція містить декілька аргументів, зберігається тип лише того аргументу, який було визначено найбільш важливим на попередньому кроці аналізу – визначення аргументів.

Таблиця 3.3 – Відповідність між узагальненими типами даних та типами даних у мові C#

Узагальнений тип даних	Тип даних мови C#
Рядок	string
Число	int, double, float, decimal, byte
Послідовність	Array, List, IEnumerable, ICollection
Тип, створений користувачем	Будь-який тип, крім вказаних вище
Невідомий тип	Тип невизначено

Якщо вираз містить декілька операндів, визначається найбільш загальний тип. Операнди, що мають невизначений тип – ігноруються. З метою спрощення та пришвидшення синтаксичного аналізу виразів виведення типів виконується за спрощеними правилами, що не завжди відповідають семантиці мови, але можуть виконуватись максимально швидко. Правила виведення типів наступні:

- операнди, тип яких невідомий – не розглядаються;
- якщо усі операнди, тип даних яких відомий, мають однаковий тип – тип даних виразу визначається відповідно до усіх його операндів;
- якщо тип даних операндів відрізняється, тип виразу визначається як тип операнду, що має найвищий пріоритет.

Пріоритет типів відповідає порядку типів у таблиці. Можливе некоректне визначення типу компенсується ефективністю роботи алгоритму, порівняно з класичним, а також тим, що наявність інших ознак забезпечує можливість визначення неповного збігу навіть при наявності помилок.

Конвертація в текст виконується за допомогою послідовного обходу трансформованого графа потоку керування в ширину та послідовним записом кожного ребра у файл (або у пам'ять). Кожне ребро характеризується парою елементів, яким воно інцидентне. Ребро записується у наступному вигляді:

- початкова вершина;
- символ об'єднання;
- кінцева вершина.

3.4. Інтерфейс користувача

Розроблена програмна система може використовуватись як у якості модуля аналізу коду для середовища розробки Microsoft Visual Studio, так і у якості самостійного програмного продукту. Система призначена для використання у автоматизованих системах пошуку дублікатів коду, з автоматичним або напівавтоматичним режимом запуску. Таким чином вимоги до інтерфейсу наступні:

- текстовий інтерфейс користувача (рис. 3.5);
- стандартизований вивід, що може бути прочитано та розпізнано автоматизованими парсерами;
- можливість передавання усіх необхідних аргументів у командному рядку;
- можливість роботи без інтерактивного режиму;
- запис усіх дій та помилок, що виникли у процесі роботи, у файли журналу з можливістю їх автоматизованого аналізу;
- наявність результатів роботи у вигляді окремих файлів текстового формату.

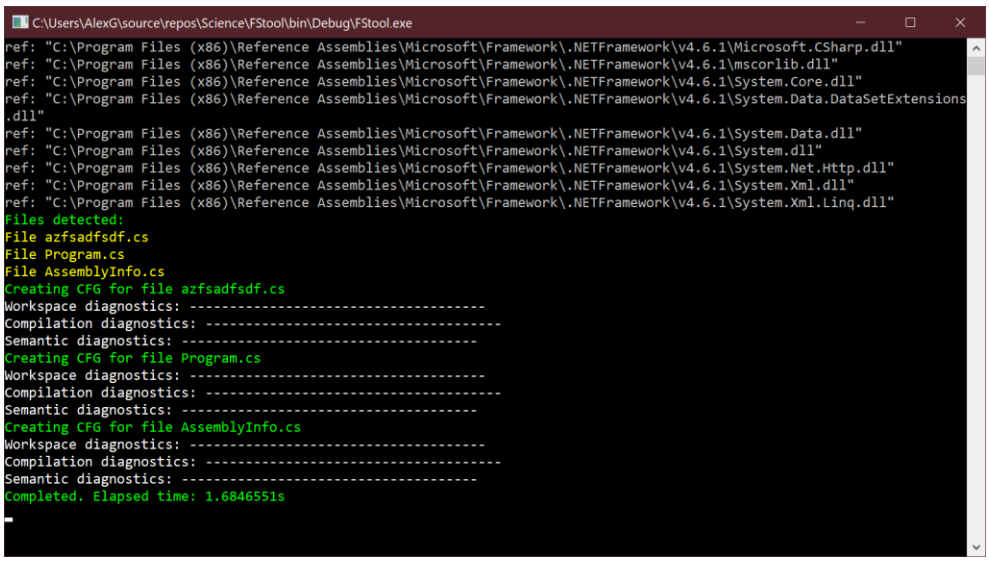


Рисунок 3.5 – Текстовий інтерфейс користувача

Основний режим роботи програми, що призначена для пошуку дублікатів коду у репозиторіях великого розміру – текстовий. Для програм текстового режиму існують такі способи передачі параметрів:

- аргументи командного рядка;
- конфігураційні файли;
- інтерактивні режим.

Так як розроблена система аналізу коду призначена для використання у складі інших автоматизованих систем, інтерактивний режим не розглядається. Аргументи командного рядка призначені для передачі параметрів, що можуть змінюватись після кожного запуску. Серед таких параметрів обов'язковим є шлях до репозиторію (або до двох репозиторіїв, якщо необхідно виконати їх порівняння), шлях для збереження файлів результату (за замовчуванням можна використовувати поточну директорію), режим роботи. У конфігураційні файли можна винести наступні опційні налаштування:

- режим роботи з файлами журналу: запис всіх дій, запис лише помилок, запис критичних помилок;
- розташування файлів журналу;
- режим роботи: пошук дублікатів, порівняння;
- мінімальні коефіцієнти подібності для ребер та графів;

- формат звіту;

Робота зі стандартним текстовим виводом: вивід всіх дій, вивід лише помилок, вивід лише критичних помилок, вивід відсутній;

- мінімальний розмір файлу для порівняння;
- кешування графів потоку керування;
- розташування збережених графів потоку керування.

Більшість з цих параметрів можуть бути встановлені як за допомогою конфігураційних файлів, так і за допомогою аргументів командного рядка. Вищезазначені параметри налаштовуються при початковій конфігурації системи та можуть використовуватись без змін.

Параметри системи описуються у файлі формату TOML, що є достатньо зручним як для читання та редагування користувачем, так і для автоматичного генерування та розбору. Приклад конфігураційного файлу наведено на рисунку 3.6.

```toml
title = "cd config"

[logs]
error = "/error.log"
rotation = true
verbose = false

[general]
mode = "detection"
similarity_k = 0.7
similarity_i = 0.8
min_file_size = 4
output = "txt"
cache = ["/home/cache", "/tmp"]
```

Рисунок 3.6 – Файл конфігурації у форматі TOML

Для використання конфігураційного файлу, його можна зберігати у спеціальній директорії (директорія, у якій знаходиться програма), або передавати шлях до конфігураційного файлу у вигляді аргументу командного рядка. За рахунок можливості передачі шляху до

конфігураційного файлу у вигляді аргументу забезпечується можливість використання різних конфігурацій для різних призначень.

Результатом роботи програми є файли звіту (рис. 3.7). Файли звіту містять повну інформацію про виконаний аналіз та знайдені дублікати коду. Склад звіту включає наступні розділи:

- інформація про запуск: дата, час, шлях до репозиторію, параметри конфігурації;
- статистика: кількість проаналізованих графів та файлів, індекс подібності;
- помилки, що виникли при обробці даних (може бути відсутнім);
- перелік файлів, що містять дублікати коду.

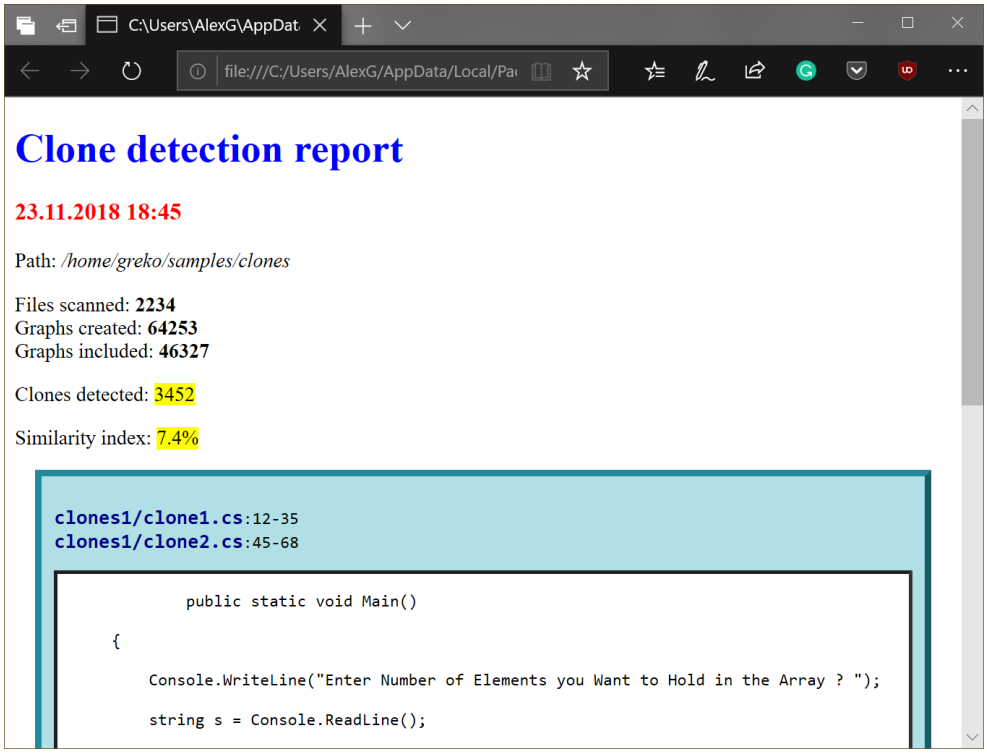
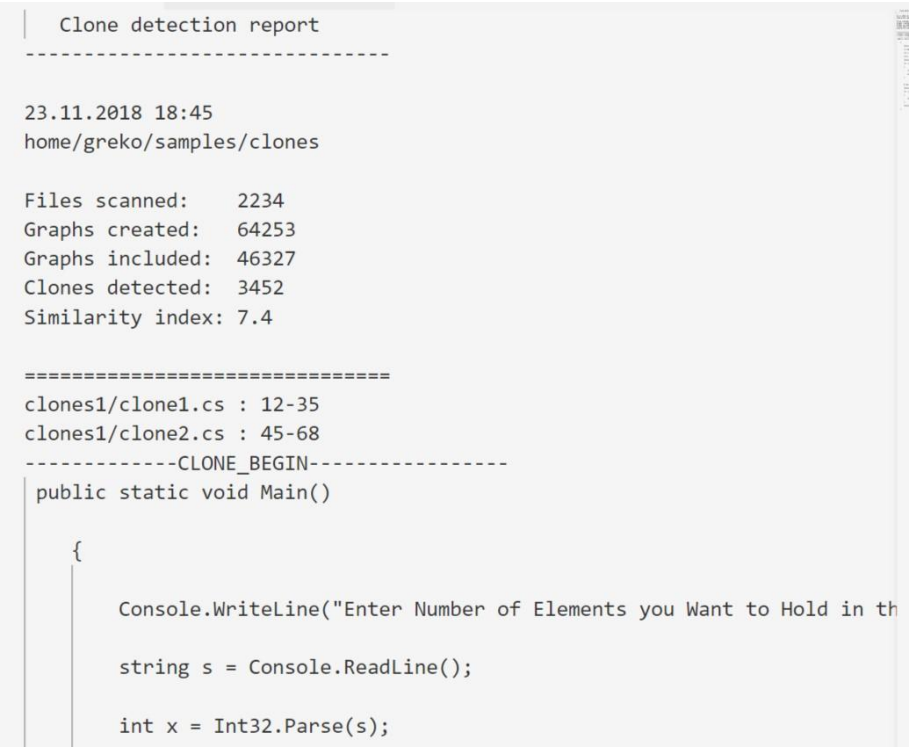


Рисунок 3.7 – Звіт у форматі HTML

Кожен файл звіту містить інформацію про кількість проаналізованих файлів, про кількість побудованих графів потоку керування (так як їх кількість відповідає кількості проаналізованих функцій та не відповідає кількості файлів), про синтаксичні, лексичні та семантичні помилки, якщо вони виникли при виконанні аналізу файлів, або про помилки програми, що можуть виникнути у процесі роботи (рис. 3.8). Кількість графів, що

відповідає умовам порівняння (більше мінімального розміру файлу, більше мінімальної кількості ребер), не завжди відповідає загальній кількості побудованих графів, тому виводиться окремо.



```
  Clone detection report
------------------------------

23.11.2018 18:45
home/greko/samples/clones

Files scanned:    2234
Graphs created:   64253
Graphs included:  46327
Clones detected:  3452
Similarity index: 7.4

==============================
clones1/clone1.cs : 12-35
clones1/clone2.cs : 45-68
-------------CLONE_BEGIN-----------------
public static void Main()

   {

       Console.WriteLine("Enter Number of Elements you Want to Hold in th

       string s = Console.ReadLine();

       int x = Int32.Parse(s);
```

Рисунок 3.8 – Звіт у звичайному текстовому форматі

Загальний коефіцієнт подібності визначається співвідношенням між графами, що були розпізнані як дублікати, та загальною кількістю графів, що задовольняють критерії порівняння. Коефіцієнт подібності можна використовувати у статистичному аналізі, для оцінки якості коду та для оцінки складності підтримки проекту. Високі значення коефіцієнта подібності однозначно вказують на необхідність виконання рефакторингу та видалення дублікатів коду. У режимі порівняння двох репозиторіїв високий коефіцієнт подібності вказує на можливе копіювання коду.

Альтернативою звітам у форматі HTML є звіти у звичайному текстовому форматі. Перевагою такого формату є можливість простого аналізу файлів звіту за допомогою програмного забезпечення та інтеграції результатів пошуку дублікатів у інші системи аналізу коду. Текстовий звіт містить ту ж саму інформацію, що і звіт у форматі HTML, але

використовує спеціальні символи для форматування виводу та розділення звіту на логічні частини.

Звіт про помилки створюється при кожному запуску програми та містить перелік помилок, а також (якщо увімкнено максимальний рівень деталізації журналів) містить інформацію про початок та кінець аналізу кожного файлу, виконані трансформації та створені файли текстового представлення графів потоку керування. Шлях до звіту про помилки відповідає шляху до виконуваного файлу програми, якщо інше не вказано у файлі конфігурації або у аргументах командного рядка.

3.5. Підхід до тестування системи

Метою тестування розробленої програмної системи є перевірка коректної роботи основних функцій, серед яких:

- пошук файлів для аналізу;
- виконання лексичного та синтаксичного аналізу;
- побудова графу потоку керування;
- трансформація графу потоку керування;
- запис графів потоку керування у відповідні файли;
- порівняння графів потоку керування;
- створення звітів різних форматів та звіту про помилки.

Для тестування першого етапу – пошуку файлів – доцільним є використання проекту на мові програмування C#, що містить складну ієрархічну структуру директорій. Файли для аналізу завантажуються за допомогою бібліотеки Roslyn, що аналізує файли опису проекту MSBuild. Для коректної роботи з кодом на мові C# необхідно мати коректні файли опису проекту. Для інших мов програмування потрібні інші файли, що відповідають системі збірки конкретної мови програмування. Аналіз залежностей проекту не є необхідним через те, що рівень семантичного аналізу, потрібний для трансформації графів потоку керування, не вимагає

повного аналізу типів програми. Для мов програмування, що не мають єдиної стандартної систем збірки, зокрема Java, C та C++, коректним способом роботи системи є рекурсивний обхід директорій, починаючи від кореневої директорії репозиторія, переданої у якості параметру.

```
digraph {
"while-stmt" -> "expression"
"while-stmt" -> "statement-list"
expression -> "=="
"==" -> dd
"==" -> "\"a\""
"statement-list" -> "dd = dd.Substring(1)"
"statement-list" -> "i++"
}
```

Рисунок 3.9 – Запис графа у форматі програми dot

Виконання лексичного та синтаксичного аналізу у розробленій системі пошуку дублікатів коду забезпечується за рахунок вбудованих можливостей бібліотеки Roslyn. Вхідними даними для аналізатору бібліотеки Roslyn є файл опису проекту, або список файлів для аналізу. Вихідними даними є побудоване синтаксичне дерево та набір повідомлень про лексичні та синтаксичні помилки. Для тестування даної частини системи необхідно використовувати синтаксично коректні, а також синтаксично або лексично некоректні файли вихідного коду мовою C#.

Побудова графу потоку керування виконується за допомогою обходу в глибину побудованого на попередньому кроці синтаксичного дерева. Для перевірки коректності створення графу потоку керування, необхідно зберігати графи у графічному форматі та оцінювати результат роботи візуально, так як будь-які автоматизовані способи перевірки неможливі. Якщо формат файлів графів зафіксовано, можливе написання автоматизованих тестів. Для візуального представлення графів використовується програма GraphViz. Програма GraphViz призначення для представлення графів, записаних у спеціальному текстовому форматі, у

вигляді графічних файлів. Формат вхідних даних - «dot», що являє собою опис вершин графа та ребер графа у текстовому вигляді. Для ефективного тестування та відлагодження програми, було розроблено можливість запису графів у даний формат, а також автоматична конвертація графів у графічний вигляд за допомогою програми GraphViz.

```
static void A3()
{
    bool sdfe = false;
    var akdt = "dfsva34g";
    while (!("a" != akdt))
    {
        {
            if (sdfe)
            {
                dfvh();
                dawe();
            }
            else
                opdj();
        }
    }
    dawe();
    if (sdfe)
        akdt = "sdfgr3er";
}
```

Рисунок 3.10 – Приклад згенерованого коду

Трансформація графів потоку керування виконується за допомогою аналізу ознак кожної програмної інструкції у кожній вершині графа. Для тестування програми реалізована можливість виведення графу після трансформації у формат, що відповідає програмі GraphViz, з подальшою його візуалізацією.

Запис графу потоку керування у відповідні файли виконується після трансформації графів та відповідає налаштуванням програми. У спеціальне місце файлової системи, що визначено параметрами конфігурації, записуються файли, що містять стиснений запис графа потоку керування кожної функції. Для тестування даного функціоналу перевіряється наявність файлів, відповідність кількості створених файлів кількості

функцій у вхідному коді, відповідність змісту кожного файлу трансформованому графу потоку керування та функції, на базі якої граф було побудовано.

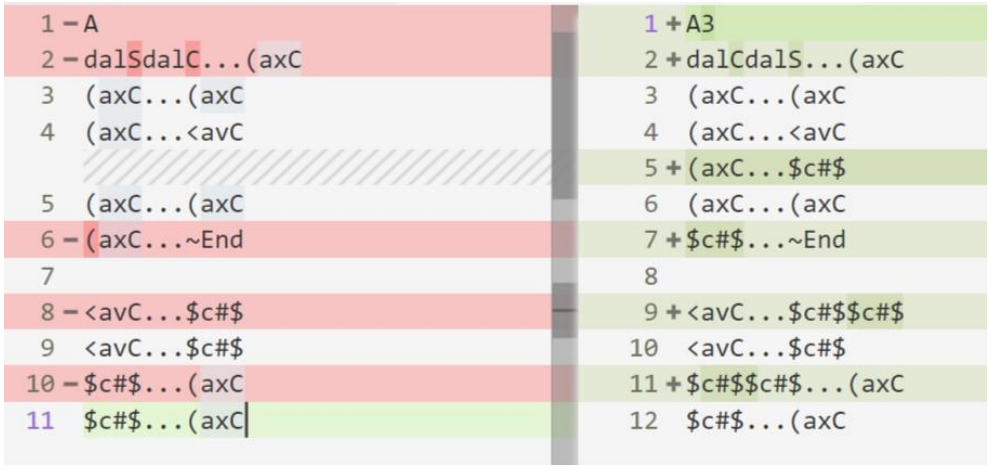


Рисунок 3.11 – Порівняння текстового представлення

Порівняння – головний етап роботи програми, так як пошук дублікатів коду базується на порівнянні графів. Для того, щоб перевірити коректність роботи даного етапу, необхідно створити код, що містить дублікати різних типів, та перевірити коректність розпізнавання дублікатів. Для порівняльного аналізу розробленого способу з існуючими аналогами, а також для тестування коректності роботи основного функціоналу програми використовується генератор коду.

Аналіз звітів на коректність виконується візуально за допомогою перегляду звітів формату HTML у будь-якому браузері, а звітів звичайного текстового формату – у текстовому редакторі. Звіт має містити усі категорії та інформацію про усі дублікати, що присутні у вихідному коді програми.

Генератор вхідних даних призначений для перевірки на коректність роботи програми, а також для створення великих об'ємів синтетичних даних для тестування та порівняння способів пошуку дублікатів коду. Основні вимоги до генератора коду:

- генерування різного коду мовою C#, що містить випадкові дії, виклики функцій, декларування та ініціалізацію змінних, цикли

та умови.

- синтаксична та лексична коректність згенерованого коду;
- можливість виконання компіляції згенерованого коду без помилок компіляції;
- можливість налаштування кількості, об'єму та вмісту файлів вихідного коду, що генеруються;
- кожен запуск генератора коду повинен створювати різні вихідні файли;
- створення коректних файлів опису проекту;
- можливість створення обмеженої кількості дублікатів першого, другого та третього типів.

Розроблено генератор вхідних даних, що створює випадковий код мовою C#. Вихідний код не має сенсу, але є коректним з точки зору граматики та семантики мови C#, тому він може пройти процедуру компіляції та бути виконаним.

Генератор коду використовує генератор випадкових чисел для створення різного коду кожен раз, проте згенерований код може містити дублікати коду будь-яких типів, і вони будуть знайдені програмами для пошуку дублікатів. Крім того, існує можливість введення умисних дублікатів з метою тестування. Генератор коду підтримує наступні налаштування:

- кількість файлів;
- кількість функцій у кожному файлі;
- приблизна кількість синтаксичних інструкцій у кожній функції;
- кількість синтетичних дублікатів:
  - тип 1;
  - тип 2;
  - тип 3;
- варіативність синтаксичних конструкцій.

Коректність генератора коду перевірена за допомогою компіляції

згенерованого коду та виконання пошуку дублікатів у згенерованому коді.

Якщо розроблена програмна система відповідає усім критеріям, описаним у цьому розділі, а також задовольняє вимоги до інтерфейсу користувача, її можна вважати реалізованою коректно та виконувати порівняння програми з іншими аналогами для пошуку дублікатів коду.

3.6. Тестування системи

Тестування розробленої програми підтверджує відповідність програми визначеним технічним вимогам.

Інтерфейс користувача відповідає вимогам до текстових інтерфейсів, конфігурації за допомогою аргументів командного рядка та за допомогою файлів конфігурації. Існує можливість редагування конфігурації за допомогою усіх зазначених способів. Зміни конфігурації впливають на поведінку програми та створений результат.

Звіт про виконання пошуку дублікатів коду містить всю необхідну інформацію для статистичного аналізу, технічну інформацію про помилки та оброблені файли, а також інформацію про кожен знайдений дублікат, включаючи файли, позицію у кожному файлі та текст функції, що дублюється.

За допомогою генератора коду було створено наступні дані:

- проект, що містить 20 файлів вихідного коду, що розташовані у складній ієрархії директорій;
- кожен файл відповідає класу мови C#;
- проект містить коректний файл опису проекту;
- кожен клас містить близько 20 методів;
- кожен метод містить від 5 до 30 рядків коду на мові C#, компіляція коду можлива;

Код програми може містити:

- виклики інших методів;

- виклики системних методів;
- створення змінних;
- ініціалізацію змінних;
- умовний оператор if;
- циклічні конструкції while, for, foreach;
- оператори continue та break у тілі циклів;
- оператор return;

Для якісного тестування розробленої системи, генератор коду має можливість створювати дублікати коду першого, другого та третього типів. Кількість дублікатів регулюється за допомогою конфігурації генератора коду.

Для генерації дублікатів коду першого типу, створюється повна копія синтаксичного дерева. Відтворення синтаксичного дерева у текстовому файлі вихідного коду виконується з незначними змінами форматування. Таким чином забезпечується повний дублікат функції першого типу. Завдяки вільним правилам форматування коду у мові C#, можна створити велику кількість різних дублікатів першого типу.

Для створення дублікатів коду другого типу створюється повна копія синтаксичного дерева, до якої додаються коментарі у випадкових місцях, а також виконується обхід дерева та заміна ідентифікаторів та констант на інші випадкові значення.

Для створення дублікатів коду третього типу також створюється копія синтаксичного дерева та виконуються усі трансформації, описані у попередньому кроці. Під час обходу синтаксичного дерева, крім описаних вище модифікацій ідентифікаторів та констант, виконується видалення, додавання та переміщення випадкових програмних інструкцій (лише у тих випадках, коли такі операції не пошкоджують семантичну коректність згенерованого коду та не викликають помилок компіляції).

Згенеровані дублікати кожного типу успішно розпізнаються програмою. Крім дублікатів коду, створених за допомогою функції

генератора коду, було розпізнано інші дублікати коду, що були згенеровані випадково у звичайному коді. За проведеним дослідженням можна зробити висновок, що програма для пошуку дублювання коду працює коректно у межах, визначених вимогами до розроблюваної програми.

4. ПОРІВНЯЛЬНИЙ АНАЛІЗ ЕФЕКТИВНОСТІ СПОСОБІВ

4.1. Аналіз способів порівняння програмного коду

Для створення системи пошуку дублікатів програмного коду на основі порівняння дерев потоку керування було обрано мову програмування C#. Метою створеної системи є тестування розробленого способу пошуку дублікатів коду. Тестування системи виконується за трьома основними критеріями:

- ефективність роботи;
- кількість знайдених дублікатів коду;
- кількість помилково визначених дублікатів коду.

Даний набір характеристик дозволяє оцінити основні властивості розробленого способу пошуку дублікатів коду. З точки зору практичної реалізації розробленого способу розглядаються наступні характеристики:

- кешування результатів роботи програми для пришвидшення роботи під час наступних запусків;
- необхідна кількість оперативної пам’яті;
- ефективність використання ресурсів комп’ютера за наявності багатоядерного процесора;
- необхідна кількість вільного місця у постійній пам’яті комп’ютера.

Для тестування способів пошуку дублювання коду використовується два набори даних: реальні та синтетичні.

Для створення синтетичних даних використовується генератор випадкового коду. Генератор вхідних даних призначений для створення великих об'ємів синтетичних даних для тестування та порівняння способів пошуку дублікатів коду. Генератор коду використовує генератор

випадкових чисел для створення різного коду кожен раз, проте згенерований код може містити дублікати коду будь-яких типів, і вони будуть знайдені програмами для пошуку дублікатів.

Структура проекту, який було створено за допомогою генератора коду, показано на рис. 4.1.

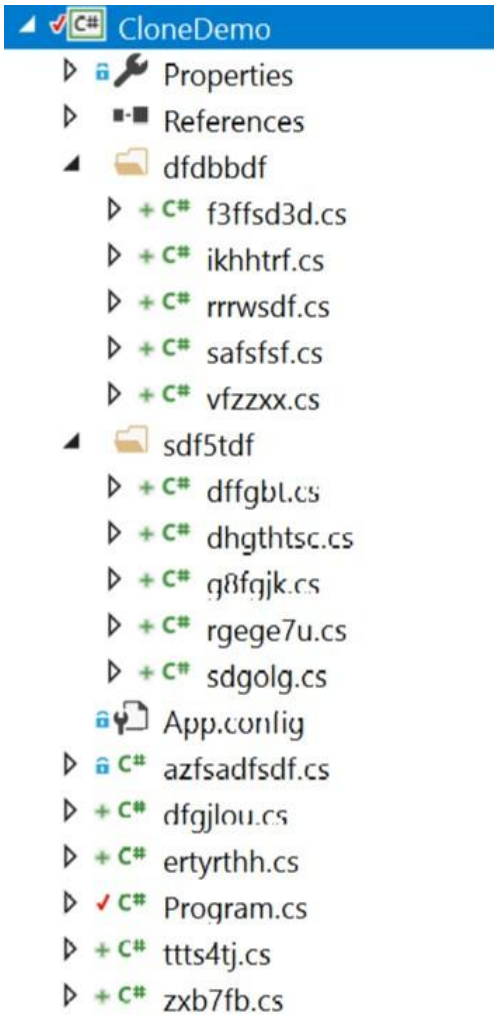


Рисунок 4.1 – Структура проекту, створеного за допомогою генератора коду

Назви класів, функцій, методів та змінних, а також значення констант – випадкові текстові рядки. Вихідний код не має сенсу, але є коректним з точки зору граматики та семантики мови C#, тому він може пройти процедуру компіляції та бути виконаним.

Так як генератор коду має можливість налаштування кількості

генерованих рядків коду, об’єм згенерованого коду можливо інкрементально збільшувати для побудови рівних графіків відповідності між часом виконання програми та об’ємом вхідних даних.

Також існує можливість створення синтетичних дублікатів коду кожного з трьох типів.

Дублікати першого типу відрізняються форматуванням коду (рис. 4.2):

- перенос на наступний рядок;
- розміщення відкриваючої та закриваючої дужки блоку;
- відступи та символи табуляції.

```
static void A3()
{
    bool sdfe = false;
    var akdt = "dfsva34g";
    while (!("a" != akdt))
    {
            if (sdfe)
            {
                dfvh();
                dawe();
            }
            else
                opdj();
    }
    dawe();
    if (sdfe)
        akdt = "sdfgr3er";
}
```

```
static void B4()
{
    bool sdfe = false;

    var akdt = "dfsva34g";

    while (!("a" != akdt)){
            if (sdfe) {
                dfvh();
                dawe(); }
            else opdj();
    }
    dawe();
    if (sdfe) akdt = "sdfgr3er";
}
```

Рисунок 4.2 – Дублікат коду першого типу, що створено за допомогою генератора коду

Дублікати другого типу відрізняються іменами (рис. 4.3), а також можуть включати усі зміни, що існують у дублікатах першого типу:

- іменування ідентифікаторів: змінних, назв методів;
- значення констант.

```
static void A3()
{
    bool sdfe = false;
    var akdt = "dfsva34g";
    while (!("a" != akdt))
    {
            if (sdfe)
            {
                dfvh();
                dawe();
            }
            else
                opdj();
    }
    dawe();
    if (sdfe)
        akdt = "sdfgr3er";
}
```

```
static void A4()
{
    bool zce4 = true;
    var hNDf = "sfgsdgsg";
    while (!("dcdgs" != hNDf))
    {
        if (zce4)
        {
            dfvh();
            arft();
        }
        else
            opdj();
    }
    arft();
    if (zce4)
        hNDf = "sdfgr3er";
}
```

Рисунок 4.3 – Дублікат коду другого типу, що створено за допомогою генератора коду

Дублікати третього типу відрізняються усіма вищезазначеними модифікаціями другого та третього типу, а також наступними модифікаціями, унікальними для дублікатів третього типу (рис. 4.4):

- модифікація виразів;
- розділення або об’єднання декларування та ініціалізації локальних змінних;
- розподіл коду на блоки, що можуть містити один елемент, або видалення блоків з одного елементу;
- видалення інструкцій;
- модифікація інструкцій;
- додавання інструкцій;
- зміна порядку інструкцій;
- зміна типів циклічних інструкцій;
- зміна типів умовних конструкцій;
- зміна типу даних;
- зміна конкретних функцій, що викликаються.

```
static void A3()
{
    bool sdfe = false;
    var akdt = "dfsva34g";
    while (!("a" != akdt))
    {
            if (sdfe)
            {
                dfvh();
                dawe();
            }
            else
                opdj();
    }
    dawe();
    if (sdfe)
        akdt = "sdfgr3er";
}
```

```
static void A33()
{
    var akdt = "dfsva34g";
    bool sdfe = false;
    while ("a" != akdt)
    {
        if (!sdfe)
        {
            opdj();
        }
        else
        {
            dawe();
            dfvh();
        }
    }
    if (sdfe)
    {
        akdt = "sdfgr3er";
    }
}
```

Рисунок 4.4 – Дублікат коду третього типу, що створено за допомогою генератора коду

Дублікати четвертого типу – це дублікати коду, що не є синтаксично подібними. Інша реалізація одного й того самого алгоритму є дублікатом коду четвертого типу. Не існує певних модифікацій, які можна було б виділити для генерування коду, який включає умисні дублікати четвертого типу. Наразі не існує жодної системи, що може визначати дублікати коду четвертого типу. В загальному випадку дублікати коду четвертого типу не вважаються дублікатами, тому їх пошук не є задачею, що розглядається у даному проекті. Генератор коду не має можливості створення дублікатів четвертого типу.

Тестування на реальних даних виконується на існуючих проектах, створених на мові програмування C#. Було обрано декілька проектів з відкритим кодом, що доступні на платформі GitHub, а також власні проекти. За допомогою тестування на реальних даних, включаючи власні програмні проекти, можливо перевірити якість роботи та можливість використання розробленого способу пошуку дублікатів у щоденній практиці звичайної розробки проектів.

У якості кандидатів для виконання аналізу коду використовуються наступні проекти:

- Shadowsocks for Windows;
- NoPowerShell;
- ASP.NET Core;
- NetworlModeller – власний проект з відкритим вихідним кодом для моделювання комп’ютерних мереж.

NoPowerShell – це інструмент, реалізований на мові програмування C#, який підтримує виконання PowerShell-подібних команд, залишаючись невидимим для будь-яких механізмів реєстрації PowerShell. Використовується лише мова програмування C# та стандартна бібліотека цієї мови, без зовнішніх залежностей. Забезпечується сумісність з .NET Framework версії 2.0. Даний інструмент може використовуватись для виконання команд у пам'яті.

Таблиця 4.1 – Об’єм тестових проектів на мові C#

Назва	Кількість рядків коду
Shadowsocks for Windows	6928
NoPowerShell	1196
ASP.NET Core	811042
NetworkModeller	2348

Shadowsocks for Windows – реалізація протоколу socks5 на платформі .NET Framework для операційної системи Windows, що дозволяє створювати захищений від прослуховування та аналізу канал зв’язку через мережу Інтернет. Підтримуються різні режими роботи, в тому числі робота в якості проксі сервера та вихідної точки. Можливе розширення функціональності за допомогою додатків, написаних на мовах

програмування C# або F#.

ASP.NET Core – веб-фреймворк від компанії Microsoft, що використовується для розробки серверної частини додатків. Підтримуються усі основні операційні системи: Linux, Windows, Mac OS. Додатки, що використовують фреймворк ASP.NET можуть запускатися на .NET Core або на повній платформі .NET Framework. Він був розроблений для забезпечення оптимізованої структури розробки для додатків, які розгортаються в хмарі або запускаються на місцевому рівні. Він складається з модульних компонентів з мінімальними накладними витратами, тому зберігається гнучкість при побудові проектів на основі даного фреймворку.

4.2. Аналіз способів порівняння програмного коду

Було проведено порівняльний аналіз наступних способів:

- порівняння на основі тексту за допомогою існуючого програмного забезпечення;
- порівняння даних на рівні графів потоку керування за допомогою класичного алгоритму, власна реалізація;
- порівняння графів потоку керування за допомогою розробленого способу порівняння графів.

Для порівняння ефективності роботи розробленого способу порівняння графів використовується власна реалізація класичного способу порівняння графів. Реалізація має наступні особливості:

- побудова класичного графу потоку керування, без модифікацій;
- використання вбудованих функцій для побудови графу потоку керування;
- відсутність трансформації графів та стисненого представлення;

- використання типових функцій серіалізації мови програмування C# для зберігання та відновлення у пам'яті побудованих графів потоку керування;
- використання однакових коефіцієнтів визначення рівності для неповних збігів, як і для розробленого способу.

Програма для тестування, що реалізує класичний алгоритм порівняння, працює наступним чином. Лексичний та синтаксичний аналіз виконується за допомогою вбудованих можливостей фреймворка Roslyn аналогічно реалізації розробленого способу. Наступні кроки алгоритму відрізняються [15].

Побудова графу потоку керування виконується за допомогою модуля семантичного аналізу фреймворка Roslyn, за рахунок чого забезпечується більш ефективна побудова класичного немодифікованого графу потоку керування для кожного метода. Ефективність досягається за рахунок використання оптимізацій, що відсутні у розробленому прототипі.

Побудований граф потоку керування зберігається у двійковий файл, що відповідає опису об'єкта у пам'яті з точки зору віртуальної машини .NET CLR, використовуючи вбудовані функції серіалізації. Запис двійкових файлів без необхідності конвертації їх у інший (текстовий) формат виконується швидше за формування та створення текстових файлів, які використовуються у розробленому прототипі. За рахунок даних факторів очікується підвищення швидкості створення та збереження графів потоку керування у власній реалізації класичному способі порівняння графів.

Зберігання графів потоку керування у вигляді серіалізованих об'єктів платформи .NET Framework вимагає значно більше пам'яті, у порівнянні з стисненим текстовим представленням. Це відбувається через необхідність зберігати додаткову інформацію про типи об'єктів, що забезпечується процесом серіалізації платформи .NET Framework, а також через те, що кожна вершина графу потоку керування без модифікацій містить повне

синтаксичне дерево (або його представлення у вигляді повного тексту) для кожної програмної інструкції.

Порівняння серіалізованих графів потоку керування відбувається інакше, у порівнянні з прототипом розробленого способу порівняння. По-перше, виконується десеріалізація графів у пам'ять з метою їх порівняння. Цей процес є досить швидким за рахунок оптимізацій, існуючих у віртуальній машині .NET CLR, проте не швидшим за читання звичайного текстового файлу значно меншого об'єму. Подальше порівняння виконується за методом пошуку підграфів у графі. Виконується обхід графів у пам'яті більше одного разу, за рахунок чого швидкість значно падає відносно посимвольного порівняння тексту, реалізованого у прототипі розробленого способу порівняння графів. Так як кількість необхідних порівнянь зростає нелінійно при збільшенні кількості методів чи функцій у вхідних даних, оптимізація процесу порівняння є основною задачею прискорення процесу порівняння графів потоку керування.

Тестування програм на згенерованих вхідних даних виконується за допомогою описаного вище генератора коду, а також за допомогою заміру часу та порівняння результатів. Отримані наступні результати, що представлені графічно на рисунку 4.5.

Генерування результатів виконується послідовно, з кроком у 100 рядків коду. На кожній ітерації кількість файлів, що генерується, збільшується на 10, а також кількість рядків коду у кожному файлі збільшується на 10. Рядки коду порівну розподіляються по методам кожного файлу (один файл відповідає одному класу). Заміри часу виконуються на комп'ютері з наступними характеристиками:

- процесор: Intel Core i7 6700k: 4 ядра, 8 потоків, максимальна частота ядра у однопоточному режимі – 4 ГГц;
- оперативна пам'ять: DDR4 з тактовою частотою 2400 ГГц, об'єм пам'яті – 16 ГБ;

- постійний запам'ятовуючий пристрій: SSD Samsung 850 EVO, що використовує інтерфейс SATA версії 3.0;
- операційна система: Microsoft Windows 10.

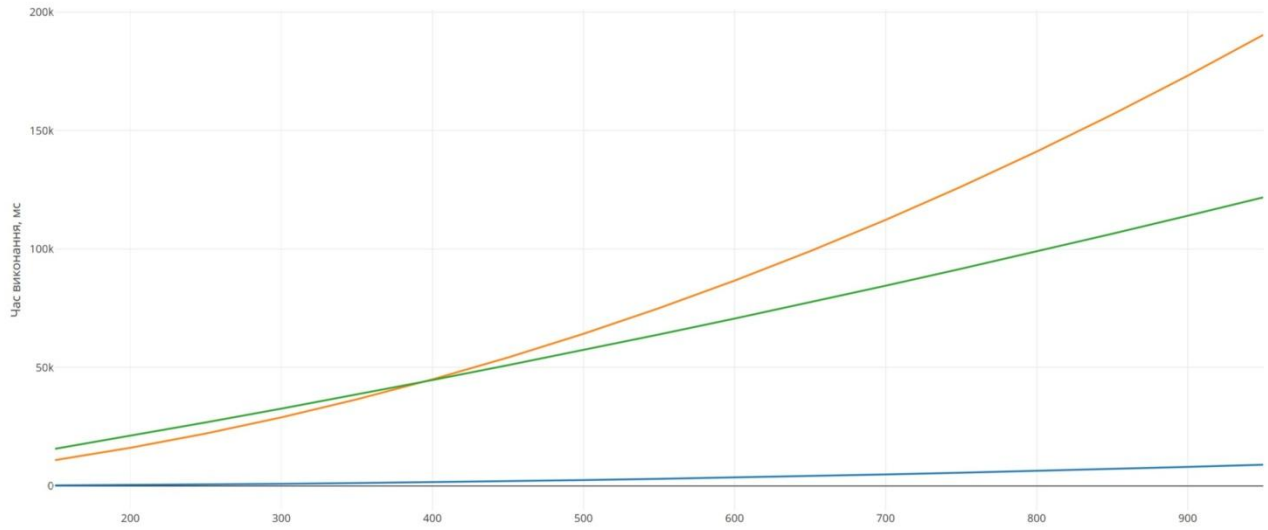


Рисунок 4.5 – Графік порівняння способів за часом виконання в залежності від об’єму вхідних даних

Інші характеристики не можуть суттєво вплинути на результат дослідження, і тому не були наведені.

Замір часу виконується за допомогою утиліти командного рядка з пакету основних утиліт операційної системи Linux – «time». За відсутності даної програми у системі Windows використовується її стороння версія з пакету «Rust coreutils», написана на мові програмування Rust.

Отримані результати повністю відповідають очікуваним теоретичним прогнозам. За рахунок того, що класичний спосіб побудови та порівняння графів потоку керування є більш ефективним при виконанні побудови та зберігання графів потоку керування – на менших вхідних даних, де кількість необхідних порівнянь незначна, загальний час роботи класичного алгоритму менший, що свідчить про його вищу ефективність за розроблений спосіб порівняння.

На вхідних даних, об'єм яких перевищує 400000 рядків вихідного коду, розроблений спосіб дає кращі результати за класичний спосіб

порівняння. Це забезпечується за рахунок значної оптимізації процесу порівняння згенерованих графів потоку керування.

Інші важливі порівняльні характеристики – це кількість та якість знайдених дублікатів коду. Кількість знайдених дублікатів – абсолютна величина, що дорівнює кількості знайдених збігів у кінцевому результаті роботи програми.

Кожна програма має можливість виведення кількості знайдених дублікатів, а розроблені прототипи класичного та нового способу порівняння графів можуть створювати файли звіту, що містять усю необхідну інформацію.

Якість знайдених дублікатів коду визначити складніше. Єдиним критерієм якості знайдених дублікатів є їх подібність. Якщо дублікати є фактичними дублікатами коду, їх визначення програмою для пошуку дублікатів є якісним. Якщо знайдені дублікати коду не є дублікатами фактично - визначається помилковий збіг.

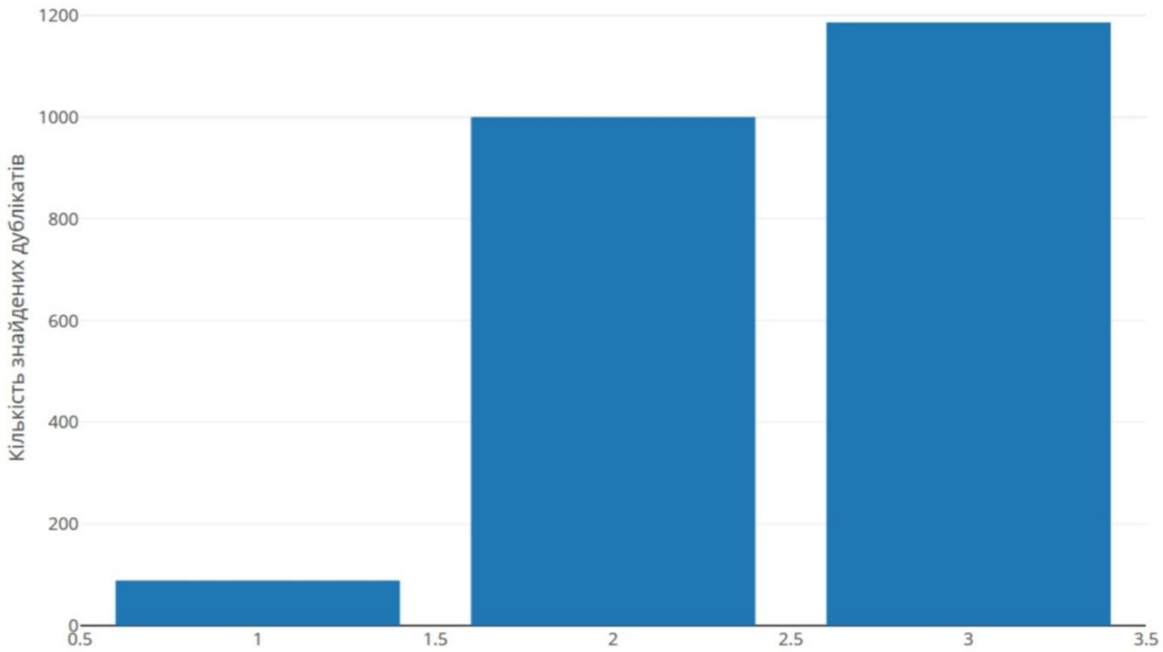


Рисунок 4.6 – Графік порівняння способів за кількістю знайдених дублікатів коду

За еталон визначення дублікатів приймемо власну реалізацію процесу порівняння графів потоку керування за допомогою класичного способу. Наведені результати тестування показують, що розроблений спосіб визначає найбільшу кількість дублікатів, більше 10% відсотків з яких є помилковими збігами. Також результати роботи розробленого способу порівняння включають 100% результатів, знайдених еталонним способом. Текстовий спосіб порівняння знаходить лише близько 20% еталонних результатів, тому що за допомогою текстових методів можна знаходити лише дублікати першого та частково другого типу.

4.3. Аналіз способів порівняння програмного коду

Крім тестування на синтетичних даних, згенерованих за допомогою генератора випадкового коду, було виконано тестування на реальних програмних проектах. Отримано результати, що наведені у таблиці 4.2.

Таблиця 4.2 – Порівняння способів на реальних тестових даних

Назва	Час виконання, класичний спосіб, мс	Час виконання, розроблений спосіб, мс	Кількість дублікатів, класичний спосіб	Кількість дублікатів, розроблений спосіб
Shadowsocks for Windows	2045	2541	12	13
NoPowerShell	958	1401	6	6
ASP.NET Core	149521	101424	148	155
NetworkModeller	1230	1864	9	10

Можна зробити висновок, що кількість фактичних дублікатів у реальних проектах менша, і це підтверджується результатами роботи обох

методів. При порівнянні кількості збігів з еталонним результатом кількість помилкових збігів на реальних даних не перевищує 10%, що відповідає визначеним вимогам до розробленого способу порівняння графів потоку керування.

ВИСНОВОК

При порівнянні запропонованого способу з звичайним способом порівняння графів потоку керування отримуємо підвищення швидкості пошуку дублікатів. Наприклад, для кодової бази проекту на мові C#, що містить близько мільйона рядків коду, побудова модифікованого графу потоку керування та його текстового представлення відбувається більш ніж на одну хвилину швидше, що відповідає прискоренню на 90% відносно класичних способів порівняння графів потоку керування програм. Такий результат забезпечується за рахунок того, що порівняння звичайних серіалізованих графів потоку керування виконується повільніше через необхідність оперувати складними структурами даних при кожному порівнянні, тоді як посимвольне порівняння текстового представлення графів виконується більш ефективно. Обидва способи тестувалися на однакових наборах даних і на одному й тому самому комп'ютері.

Порівняння представленого способу з іншими способами, що виконують порівняння на рівні тексту вихідного коду або на рівні лексем, не є доцільним, так як дані способи не можуть використовуватись для порівняння програм на різних мовах програмування та не забезпечують достатньої якості, проте гарантують значно більшу швидкість.

Слід зазначити, що представлений спосіб дозволяє знайти 100% дублікатів, які можуть бути знайдені шляхом простого порівняння графів потоку керування, включаючи дублікати першого, другого та третього типу, проте близько 10% відповідностей, знайдених даним способом є помилковими через те, що частина інформації втрачається при конвертації графа у стиснений текстовий вигляд. Також спосіб дозволяє зберігати інформацію про велику кількість графів за рахунок невеликого розміру стисненого текстового представлення графів та можливості подальшого стиснення інформації до 80% за допомогою класичних алгоритмів стиснення.

У подальшому було б доцільно узагальнити цей спосіб для інших типів графів для підвищення точності.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Фаулер М. Рефакторинг: улучшение существующего кода. — Пер. з англ. — Символ-Плюс, 2003. — 432 с
2. Юрий Карпов. Теория и технология программирования. Основы построения трансляторов — ВНУ, 2005 — 137 с.
3. Зыков, А. А. Основы теории графов — Наука, ГРФМЛ, 1987 — 384 с.
4. J.R. Ullmann, An Algorithm for Subgraph Isomorphism — National Physical Laboratory, England, 1976.
5. F. E. Allen. Control flow analysis - Proceedings of a symposium on Compiler optimization, 1970.
6. *S. Horwitz.* The use of program dependence graphs in software engineering - Proceedings of the 14th international conference on Softwareengineering, ICSE – 1992.
7. R. Sethi, J. Ullman. Compilers, Principles, Techniques and Tools – Addison-Wesley, 1986.
8. R. Frost, R. Hafiz. Parser Combinators for Ambiguous Left-Recursive Grammars - 10th International Symposium on Practical Aspects of Declarative Languages (PADL) – 2008.
9. S. Graham, M. Wegman. A fast and usually linear algorithm for global data flow analysis - Journal of the ACM – 1976
10. I. Baxter, A. Yahin, L. Moura, L. Bier, Clone Detection Using Abstract Syntax Trees - Software Maintenance, 1998.
11. J. Ferrante, The program dependence graph and its use in optimization - ACM Transactions on Programming Languages and Systems, 1987.
12. J. Ullman, J. B. Kam, Global Data Flow Analysis and Iterative Algorithms - Journal of the ACM, 1976.
13. M.Ilyas, A. Bilal, A Comparative Analysis of Clone Detection Tools: Solid SDD and CCFinderX - International Journal of Computer

Applications, 2016.

14. S. Patil, S. Chaundhari, A. Somawane, Code Clone Detection Using Hybrid Approach - International Journal of Engineering Research in Computer Science and Engineering, 2017.
15. M. Vujosevic-Janici, M. Nicolic, Software Verification and Graph Similarity for Automated Evaluation of Students' Assignments - Information And Software Technology, 2013.